INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Testbed Development and Prototypical Implementation of the Multi-Paradigm Location Language MPLL

Roman Roderick Flammer

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach

Betreuer: Dipl.-Inform. Bernhard Lorenz

Abgabetermin: 16. Januar 2007

Zusammenfassung

Innerhalb der letzen fünfzehn Jahre nahm die Bedeutung von geographischen Informationssystemen immer mehr zu. Während diese früher nahezu ausnahmslos für militärische Zwecke Anwendung fanden, profitiert man heute massgeblich sowohl im kommerziellen als auch im privaten Bereich an diesen Technologien. Diese Arbeit dokumentiert die prototypische Implementierung der Multi Paradigm Location Language, MPLL. Sie baut auf der Geo-Temporal specification language (GeTS) auf. Der Anwendungsbereich von GeTS bezieht sich auf das zeitlich basierte Schliessen. Es beinhaltet Datenstrukturen and zugehörige Funktionalität, um Zeitpunkte, Zeitintervalle und zeitliche Unterteilungen zu definieren und hand zu haben. Der Zweck von MPLL is es, das Anwendungsfeld auf zwei oder mehrdimensionale Bereiche, nämlich räumliches Schliessen, zu erweitern. MPLL dient als Kernsprache, welche eine einheitliche Grundlage für Geographische Informationssysteme (GIS) bieten soll. MPLL soll den Austausch, die Analayse, Speicherung und Manipulation von räumlichen Daten zwischen diesen Systemen ermöglichen. Die Ausarbeitung konzentriert sich auf die Implementierung von Sprachkonstrukten und Datentypen, welche ausgereiftere Herangehensweisen bezüglich ihrer Datenhandhabung ermöglichen.

Abstract

The importance of Geographic Information Systems has been growing throughout the last fifteen years. While they had almost exclusively been used for military purposes, business and private sectors now greatly profit from these technologies. This work documents the prototypical implementation of the Multi Paradigm Location Language, MPLL. It is build on the foundation of the Geo-Temporal specification language (GeTS). The area of application of GeTS is temporal reasoning. It provides data structures and functionality to define and handle points in time, time intervals, partitioning of time lines. The purpose of MPLL is to broaden the scope to a two or more dimensional subject, namely spatial reasoning. It serves as a core language that aims at providing a common ground for miscellaneous Geographic Information Systems (GIS). MPLL is supposed to allow the exchange, analysis, storage, and manipulation of spatial data among those different systems. The thesis focuses on the implementation of language constructs and data types enabling more sophisticated approaches to data handling.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 16. Januar 2007

Unterschrift des Kandidaten

Contents

List of Figures

Chapter 1

Introduction

Spatial reasoning is becoming more and more import. Todays technological advancement enables even access to systems such as route planners for the private sector. Geographic information systems (GIS) are capable of storing, modifying, analyizing, forwarding, sharing and displaying spatial information. Most GIS use their own data structures and focus on a specific domain of application. Some focus on graph specific topics such as route planning, which recently has become integrated in many applications or even mobile devices. Other systems may help to determine suitable locations for businesses to address a big amount of potential customers. Likewise they may help classifying areas which might be of climatic or geological interest. Factors like rainfall, pollution, temperature, furtility, or the existance of rural, industrial, and residential zone may be of concern. But also issues of smaller scale can be the focus of such systems. There are prototypes of navigation systems, which not only take static environment such as the course of roads, their crossings, landmarks, and obstacles into account, but also other moving vehicles or pedestrians. For such considerations certain relations between the involved entities can be determined. These are intrinsic relations, such as north, east, or west, as well as relative indications, like left, right, close, far, or even "in front of", "behind of", and "between of". For instance, the notion of the sentence "there is a child in front of the car" could imply (a) that the child is close to the car and (b) that the front side of the car approximately points into the direction of the child. Another interpretation would be that (a) the child is close to the car and (b) the child is positioned in between the car and the speaker, who made that statement. Such relationships are of qualitative nature. Deriving such relationships from a given set of informa-

tion, is often preceeded by qualitative calculations and considerations. This is part of spatial reasoning.

Due to the cirumstance that most GIS specialize in certain fields of interest, and usually do not interact between each other, the idea of MPLL emerged. It stands for "Multi Paradigm Location Language". The idea is to create a common way of exchanging spatial data as well as delegate queries to suitable systems for further processing. During this work the focus is mostly on two-dimensional spatial aspects. A former project, the "Geo-Temporal specification language" (GeTS)[1], provided this for temporal entities. These include points in time, time intervals, or partitioning of time lines. For that reason the GeTS framework is being reused to provide a foundation, on which MPLL can be implemented. As such MPLL can be seen as the transition from a one-dimensional domain to a multi-dimensional one. With a system like MPLL in place it is imaginable to merge both frameworks, in order to realize means of spatial and temporal reasoning. From a linguistic point of view, it can be noted that spatial reasoning often incorporates temporal aspects. For instance, in the sentence "The ocean is in five minutes walking distance from the hotel." the word "distance" is not expressed by actual linear measurements. In order to determine the actual distance time frame and average walking speed need to be taken into account. Likewise a distance could be translated to such a spatial-temporal expression.

This work first gives an overview of geometric primitives such as point, angles, and polygons which correspond to basic types specified by the MPLL specifcation[2]. In the next chapter the implementation is discussed. These include the introduction of said types as well as a few improvements of the framework. The latter add new features such as list types, generic types and templates to provide more comfortable and advanced means of handling the MPLL language. Additionally an application has been implemented to manage a standard library for MPLL data types and functions. It also accommodates the support for a test suite. During this work technical details are barely discussed. This is compensated by a short guide found in appendix A. It serves as a tutorial for future implementations of MPLL. It also aims at helping the reader to understand the technical aspects that had been left out during the main chapter.

Chapter 2

Overview

In this chapter an overview over the several terms from the field of geometry as well as their pendant in MPLL is recapitulated. Additionally several useful and common mathematical formulas and algorithms are provided to serve as guideline for their implementations in MPLL. In general the MPLL specification[2] is assumed to be known to the reader.

2.1 Geometric types

In the following subsections some geometric terms, their corresponding MPLL specification (if one exists) and some algorithms or mathematical principles are summarized. This helps to avoid unnecessary attention to such details in the next chapters, and allows us to divert the focus on the key points.

2.1.1 Angle 

Angles are denoted in Greek letters α, β, γ, etc. They are one the most basic as well as most essential data structures when it comes to computational geometry. They fulfill several directional purposes. Firstly, they may serve as a way of expressing the heading of a geometric entity or its direction of an intended movement. For example, a line segment may for the purpose of certain considerations be looked at as directed and thus implicates a direction. Same accounts for a point that has been added the attribute of a heading. In that case it is no longer called a point but a *configuration* which is elaborated on in 2.1.3. Secondly, angles form the means of expressing the

bearing of one entity to another. This is the case, for instance, when considering the relationship between two points but also when contemplating a point in respect to a polygon or even two polygons as illustrated in 2.1.6.7. This is elaborated in the following sections.

While angles can be described by different measurements such as degree, radian, or grad the actual entity does not rely on such ambiguous representations. Therefore the presentation of the value of an angle depends on the individual view while the internal representation may be an arbitrary but fixed one.

The same way there are ambiguous measurements that allow the representation of the same angle there are also different intervals these values may lie in. For most purposes it is insignificant whether an angle is described as $-\frac{\Pi}{2}$ or $3 \cdot \frac{\Pi}{2}$. Therefore the output is normalized in the sense, that it is restricted by a lower and upper bound. Furthermore the MPLL specification requires to provide such a minimum and maximum value when initializing an angle. It may be argued that information is lost when normalizing an angle or restricting its representation to a certain range. This is not the case when relying on trigonometric functions like sin, cos or tan nor when contemplating the final heading of an entity resulting from several full rotations. On the contrary, it is not irrelevant when the implicated rotation has a dynamic character. While this becomes more important when operating in a spatial-temporal domain, it is not to be neglected for pure spatial considerations either. It may very well be of importance to make a distinction between a rotation of 90° to the left or 270° to the right.

The MPLL specification usually contains one default constructor for each type. In the case of `Angle` it is this:

`Angle ( Fangle , Fmin , Fmax ) : Float * Float * Float` $\mapsto$ `Angle`

Further constructors rely exclusively on this default constructor such as the empty constructor for instance:

```
Angle() = Angle(0, -defMod(Grd), defmod(Grd))
```

For further information on that see standard library (see B) as well as the MPLL specification[2].

2.1.2 Point 

Points are denoted in lower-case letters p, q, etc. It is common to specify a point by writing, for example, (5|3). But this is the case when communicating points in an Euclidean space. It is the notion of MPLL to focus on geospatial data which as a corollary relies on a spherical representation. Therefore MPLL depends on the WGS84 coordinate system by default. As a consequence points in MPLL are two dimensional by nature. As a result a point is expressed as a pair of angles which reflect latitude and longitude. Spherical geometry is less trivial than Euclidean geometry. However, the intended scope of geometric considerations in MPLL focuses on rather local aspects. Therefore it suffices for the purpose of MPLL to rely on euclidean geometry for now. It is left to future extensions and modifications of MPLL to address spherical geometry.

The default constructor in the MPLL specification is defined as:

`Point ( Ax , Ay ) : Angle * Angle` $\mapsto$ `Point`

2.1.3 Configuration 

Configurations represent a position of a geometric entity as well as its orientation and possibly additional attributes. In accordance to the MPLL specification a configuration only denotes a position and an orientation. Future implementations may address the possibility to further specify such additional attributes.

The MPLL default constructor is defined as:

`Configuration ( A , Ax , Ay , Bo ) : Angle * Angle * Angle * Bool` $\mapsto$ `Configuration`

The first value denotes the orientation while the second and third angle represent the coordinates (see 2.1.2). The last parameter expresses whether the configuration is supposed to have an orientation or not. The notion here is that the orientation is deemed to be irrelevant or otherwise ignored.

2.1.4 Lines 

Lines were not specified in MPLL but may so in the future as they are frequently used data structures in the field of computational geometry. There are three different kinds of lines. A distinction is made by referring to them as *line segments*, *half lines* and *lines*. It is important to stress the difference. Not only do they differ in their possible underlying implementations but their applications vary as well.

Definition 1 (Line)**.**
A line, or straight line, can be described as an infinitely thin, infinitely long, perfectly straight curve. From now on, a *line* stands for such a straight line and is never used synonymously for a *line segment* or a *half line*. As exactly only one line contains two given non-congruent points, a line is defined by such two reference or anchor points. It is denoted as a pair of points $\overline{pq}$ where p and q serve as said anchor points.

Definition 2 (Half line)**.**
A half line or ray is part of a line bounded by one point. It consist of a starting point and from there on infinitely delates into one direction. There are two reasonable possibilities of describing a half line. One is a pair of points where the first is said starting point while the second serves as an anchor point in the sense that it is different from the first point and lies on the half line. The other possibility is to define it by a single configuration. Its location serves as the starting point and the associated orientation as the direction of extension. It is common to use the first notion and, thus, a half line is denoted as $[pq$ where p is the starting point and q an anchor point.

Definition 3 (Line segment)**.**
A line segment is part of a line bounded by two end points(wikipedia). Here it is defined as a pair of points denoted as $[pq]$ where p and q are said end points.

Geometric Algorithms

In respect to the geometric types that have been introduced so far there are a few possible relations between them. Before addressing this the introduction of some tools is useful. A huge number or problems in geometry can be solved by the use of the vector product or cross product.

Definition 2.1.4.1 (Vector Product)**.**
The vector product $\vec{a} \times \vec{b}$ of two vectors $\vec{a}$ and $\vec{b}$ in a three-dimensional Euclidean space is a vector which is orthogonal to the plane spanned by $\vec{a}$, $\vec{b}$. It is easiest described by a determinant.

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = det \begin{pmatrix} \vec{e_x} & \vec{e_y} & \vec{e_z} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix}$$

Its length is equal to the area of the parallelogram they span. The latter is the reason why the vector product is one of the most useful tools as the following definitions show. In particular the area of a triangle $(\vec{a}, \vec{b}, \vec{c})$ can be defined as:

$$area_{\Delta}(\vec{a}, \vec{b}, \vec{c}) := \frac{1}{2} \cdot |\vec{x} \times \vec{y}| \quad \text{where } \vec{x} := \vec{b} - \vec{c},\ \vec{y} := \vec{c} - \vec{a}$$

It shall be noted that the area is positive if and only if the vertices are aligned counter clockwise. As a consequence the vector product and especially its sign can be used to determine relationships between points and other entities that are defined by them as the following definition demonstrates.

Definition 2.1.4.2 (Left, Right, Colinear)**.**
Let $\vec{a}$, $\vec{b}$ and $\vec{c}$ be vectors.

$$inLine(\vec{a}, \vec{b}, \vec{c}) = |(\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})|$$

$$isLeft(\vec{a}, \vec{b}, \vec{c}) = \begin{cases} true & inLine(\vec{a}, \vec{b}, \vec{c}) > 0 \\ false & otherwise \end{cases}$$

$$isRight(\vec{a}, \vec{b}, \vec{c}) = \begin{cases} true & inLine(\vec{a}, \vec{b}, \vec{c}) < 0 \\ false & otherwise \end{cases}$$

$$isColinear(\vec{a}, \vec{b}, \vec{c}) = \begin{cases} true & inLine(\vec{a}, \vec{b}, \vec{c}) = 0 \\ false & otherwise \end{cases}$$

The predicates $isLeft$, $isRight$ are to be understood in the way whether $\vec{c}$ is left or right of the line segment starting at $\vec{a}$ and ending at $\vec{b}$.

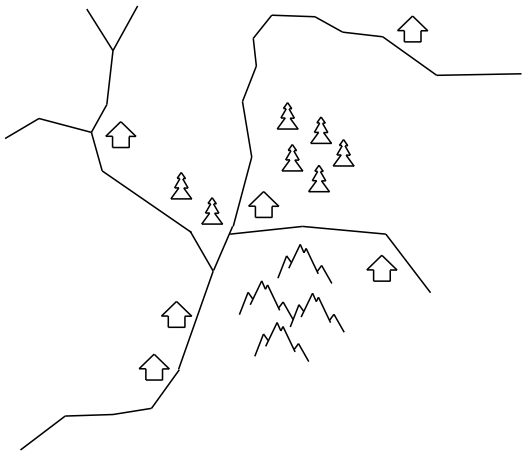


Figure 2.1: Symbolic map of a road and landmarks such as building, forrests, or hills

2.1.5 Multiline 

Definition 4 (Multiline)**.**
A *multiline* is a sequence of connected line segments. It is defined as a sequence of vertices $(v_0, \ldots, v_n)$. If first and last vertex are congruent it is called a polygon (see 2.1.6).

Even though multilines were not included in the MPLL specification their introduction is sensible. Not only has a multiline much in common with a polygon it is also applicable for several purposes. Multilines may represent a common border of two areas, such as countries, or polygons for that matter. They may also resemble roads when a pure graph theoretical representation does not suffice. This could be the case when landmarks of some sort need to be taken into account, e.g. the creation of a direction or the determination of a scenic route, as depicted in fig. 2.1.4.

2.1.6 Polygon 

Definition 5 (Polygons)**.**
A Polygon is a closed planar graph. It can be defined by a sequence of vertices $(v_0, \ldots, v_n)$ where first and last element are identical. From now on a sequence of vertices describing a sequence of line segments is referred to

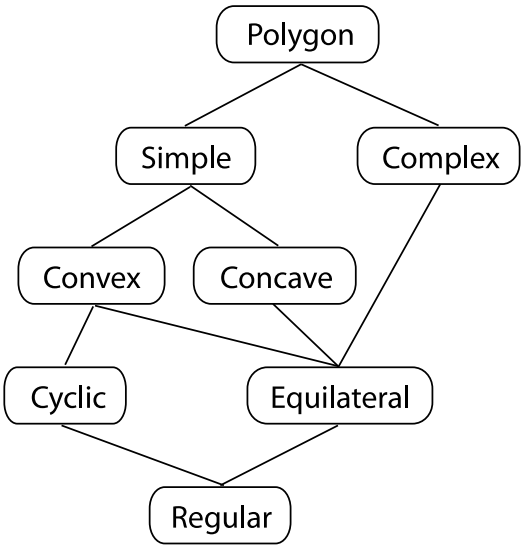


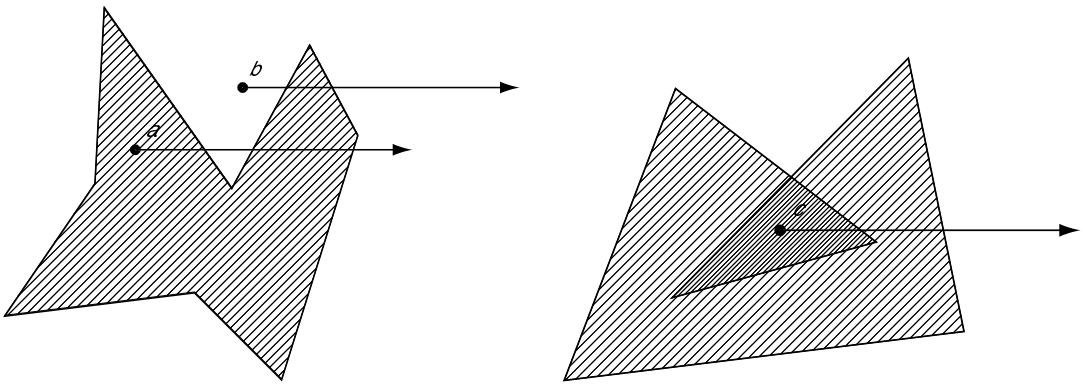
Figure 2.2: A taxonomical presentation of different categorized of polygons

as a *multiline*. Thus a polygon can be seen as a closed multiline. The line segments of a polygon are called edges. An edge is represented by its pair of vertices $(v_i, v_{(i+1) \bmod n})$.

It is common to *normalize* polygons which means the intended order of the vertices are *counterclockwise*. That can be better visualized as the immediate area to the right of an edge to be *outside* while the immediate area to the left is *inside*. More accurately a normalized polygon always has a positive area (see definition 2.1.4.1). A polygon is said to be *simple* if no edge intersects with any other edge but itself. Otherwise a polygon is said to be *complex*. From now on a polygon refers to any normalized polygon unless specified otherwise. Occasionally the term *region* is used synonymously to emphasise the area described by the polygon.

Moreover a polygon can be *convex* or *concave*. It is convex if every internal angle is at most 180 degrees. While it is possible to construct a complex polygon that matches this characteristic, complex polygons are usually neither referred to as being concave nor being convex.

There are further terms expressing possible attributes of polygons. Even though they are rarely or never used, they are mentioned for the sake of completeness. A polygon is called *concyclic* if all vertices lie on one circle



(a) Point a is in the simple polygon with $\omega = 1$. Point b is outside with $\omega = 0$

(b) Point c being wrapped twice by a complex polygon with $\omega = 2$

Figure 2.3: Inclusion of points in polygons

and *equilateral* if all edges are of the same length. If it even fulfills both criteria it is called *regular*.

Algorithm 2.1.6.1 (Winding number)**.** One of the most common algorithms to determine whether a given Point is inside or outside of a polygon is based on the *Jordan Curve Theorem*. A point is inside a polygon if the number of intersections of any ray starting at the point with the polygon is uneven; Otherwise it is outside. As every possible crossing of the ray with every line segment of the polygon has to be considered such an algorithm is in $\Theta(n)$.

There is an improved version of this algorithm which can even be applied to complex polygons and which bears a more significant result then a simple yes/no answer. This algorithm does not simply count the number of crossings it also takes the relation between a line segment and point into consideration. A line segment (l_i, l_{i+1}) *wraps* the point p if $Left(l_i, l_{i+1}, p)$. If $Right(l_i, l_{i+1}, p)$ then the line segment is said to *unwrap* the point. First a so called winding number ω is initialised with the value 0. Every time a line segment *wraps* the point the winding number is increased by one. If it *unwraps* the point the number is decreased. If the result bears $\omega = 0$ the point is outside the polygon. If $\omega > 0$ it is inside. Note that ω cannot possibly be less than zero. Obviously this algorithm has the same complexity ($\Theta(n)$). Now if the point is inside a complex polygon then ω is the number of times the polygon enclosing the point.

Algorithm 2.1.6.2 (Point in Convex Polygon test)**.** For most problems a convex polygon is much easier to handle than an arbitrary polygon. Moreover many concave polygons are often approximated by their convex hulls the following algorithm is quite useful. As it is its own convex hull (obviously) every line segment formed of arbitrary vertices of the polygon are either part of its hull or diagonals. This property can be used to formulate a point in polygon test in $\Theta(\log n)$. Essentially it is a Divide and Conquer algorithm. Let $(l_0, \ldots, l_n, l_0)$ be the polygon.

The divide step consist of splitting the polygon in two halves $(l_0, \ldots, l_{n/2})$ and $(l_{n/2}, \ldots, l_0)$. Then the relation between the line segment $(l_0, l_{n/2})$ and point p is considered. If $Left(l_0, l_{n/2}, p)$ then p has to be in the first half if it is inside the polygon at all. Otherwise in the other half. Then the same problem is recursively considered for the respective half until the remaining polygon forms a triangle. In that case a *point-in-triangle* test suffices to determine whether p is inside the polygon. As the size of the problem is reduced by half in every step ($\mathcal{T}(\backslash) = \mathcal{T}(n/2)$) the complexity is $\Theta(\log n)$.

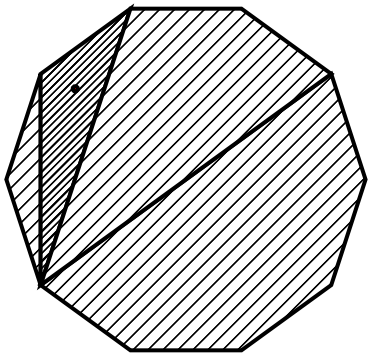


Figure 2.4: Inclusion test in regard to a point in a convex polygon.

Algorithm 2.1.6.3. [minimal bounding box] As already mentioned concave polygons are sometimes approximated. An even more common and convenient way of doing so are minimal bounding boxes (MBB). They are the smallest possible rectangles that still contain the polygon. Operating with rectangles simplifies the matter as most algorithms based on them are trivial. Furthermore it often suffices to consider the width and height and X and Y values separately. As the edges are straight lines and not arbitrary curves it

is sufficient to take only the vertices into consideration when construction a MBB. For all vertices the minimum and maximum value of each axis needs to be determined.

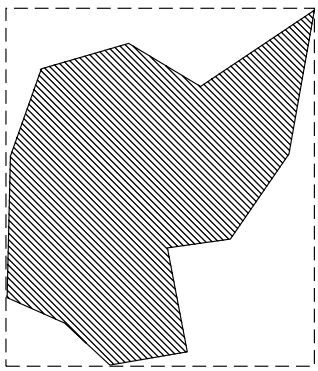


Figure 2.5: Minimal bounding box of a polygon

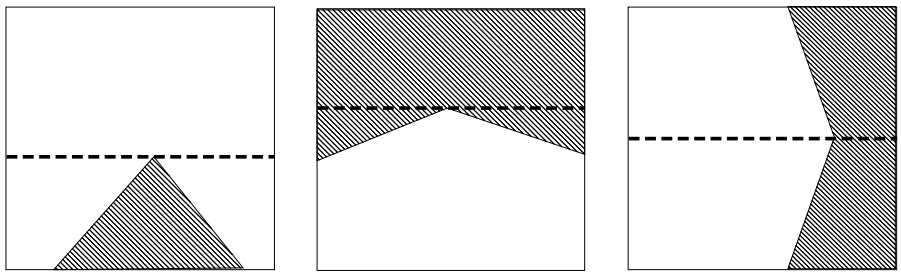
While the area of a polygon may be defined as the sum of triangles of its triangulation an algorithm using this approach would have a complexity of $\Omega(n \cdot \log(n))$ due to the process of computing the triangulation in the first place. It has been proven that the triangulation of a polygon can be done in $\Theta(n)$ in an article by Bernard Chazelle[3]. The proof is a constructive one. Unfortunately, its realization in form of an algorithm seems to be impractical due to its complexity.

Algorithm 2.1.6.4 (triangulation)**.** The decomposition of Polygons into triangles is called *triangulation*. The advantage of triangulating a polygon lies in the fact that triangles are a much easier structure to work with. As mentioned before their area can be easily calculated by using cross products. Therefore the area of a polygon is simply the sum of the triangles it is composed of. The center of mass of a polygon can be determined in a similar way which is described by the following definition.

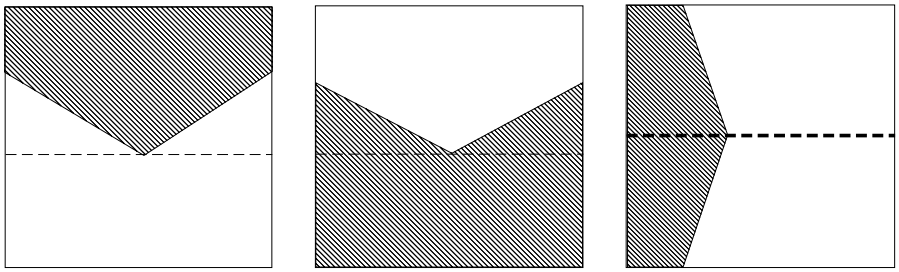
There are are several algorithms to achieve said decomposition but one of the most efficient ones is utilizing a sweep line. Its purpose is to decompose the polygon into several monotone polygons. These can be easily triangulated in linear time [4].

All vertices need to be sorted along a specific axis, e.g. along their y-values. In addition a list of all edges that intersect the scan line needs to be maintained. They are ordered by the x-values of their first intersection. The scan

line stops at every vertex where it gets categorized. Fig. 2.6 illustrates the different scenarios.



(a) Start node (b) Merge node (c) Ignore node (downward)



(d) End node (e) Split node (f) Ignore node (upward)

Figure 2.6: Six different types of node classifications. The scan line is assumed to move from the bottom to the top

The current state of intersecting lines are updated accordingly. In case of a start node the two edges that adjacent to the vertex are added. Likewise they are removed if an end node is hit. In case of a ignore node the edge is replaced by the new one. More interesting are merge and split nodes as they both entail a cut. If a merge node is encountered a cut is added as soon as the scan line stops at the next suitable node. By "suitable" it is meant that both nodes need to form a diagonal, which implies that it may not intersect any other edges. There is no need to check every other edge for a possible intersection. It suffices to look at the index position of the vertex in the list and compare it to the positions of the left and right edges of the former merge node. On the contrary a split node entails a cut with a

previously scanned node. In practice this can be realized in two ways. Either by backtracking or a second scan in opposite direction. The former requires a data structure to store not only the current state of intersecting edges but previous states as well. The latter requires a second scan to start from the opposite direction. The advantage of this approach that it simplifies the algorithm because split nodes can be simply ignored. This algorithm has a run-time complexity of $O(n \cdot \log n)$ for sorting the vertices and $O(n)$ for a complete scan sweep. Therefore the resulting complexity remains in $O(n \cdot \log n)$. The decomposition of montone polygons can be done in linear time using a stack [4].

Definition 2.1.6.5 (Centre of Mass of a Polygon)**.** The *center of the mass*, also called *centroid*, of a polygon is the arithmetical mean of the sum of the centres of masses of all triangles obtained by a triangulation. The latter can be computed fairly easy. It is the intersection of its medians which are the line segments of a vertex to the centroid of the opposite face. Let the triangle $\mathcal{T}$ be given by three vertices $\vec{a} := (x_a, y_a)$, $\vec{b} := (x_b, y_b)$, and $\vec{c} := (x_c, y_c)$. As the centroid of a d-simplex divides each median in a d:1 ratio (as seen from the vertex) and thus in a 2:1 ratio for a triangle. The function μ_∇that determines the centroid of a d-simplex $\mathcal{S}$ can be defined:

$$\mu_\nabla(S) := \frac{1}{d+1} \cdot \sum_{\vec{x} \in \mathcal{S}} \vec{x}$$

Thus for the 2-dimensional case of a triangle $\mathcal{T}$:

$$\mu_\nabla(T) = \frac{1}{3} \cdot \left(\vec{a} + \vec{b} + \vec{c}\right)$$

Now Let Δ be the set of triangles of an arbitrary triangulation of polygon $\mathcal{P}$. As mentioned above the arithmetic mean of its triangles define the centre of mass:

$$\mu(P) := \frac{1}{|\Delta|} \cdot \sum_{\delta \in \Delta} \mu_\nabla(\delta)$$

Note that the same function can be applied to the arbitrary decomposition of a polytope into a set of tetrahedra.

Definition 2.1.6.6 (Closest segment to a point)**.** Let L be a sequence of line segments and p a point. It is possible to obtain the distance between a point and a line segment by a function. It can be used to collect all closest

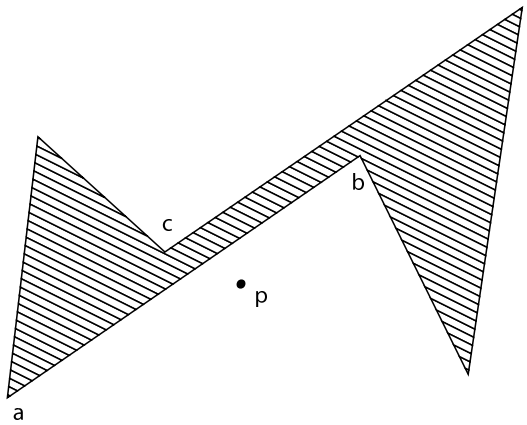


Figure 2.7: The closest line segment to Point p is $[ab]$ eventhough c is the cloest point to p.

segments of a line sequence. Note that there are several segments that may have the property to be nearest neighbour to a given point. The candidates can be defined as followed:

$$closestSegment\,(p, S) := \{s \in S | dist(p, s) \leq dist(p, x)\}$$

Note that it does not suffice to consider the distances between p and the two defining point of a line segment. Fig. 2.7 illustrates this.

Definition 2.1.6.7 (Bearing between a point and a region/sequence of line segments)**.** Let p be a point and L be a sequence of line segments. The bearing from p to L can be casually circumscribed as the part of the field of vision from p that contains L as illistrated in fig. 2.1.6 (a). It can be represented by an interval of angles as shown in fig. 2.1.6 (b). If L represents a region then the bearing interval includes even the bearing from p to every point in that region. It is evident that one interval is enough to represent the bearing as the segments are directly or indirectly connected. It is also evident that the union of intervals of the segments results in the intended bearing. Let $l := (\vec{a}, \vec{b})$ be a line segment. There are two candidates for an interval representing the bearing from a point p to l which are $[bearing(\vec{a}), bearing(\vec{b})]$ and $[bearing(\vec{b}), bearing(\vec{a})]$. Only one of them is the actual bearing while the other is the inverse interval. Obviously the bearing cannot be more than 180°. Furthermore a bearing of 180° is only possible if p lies between $\vec{a}$ and

$\vec{b}$ in which case it is ambiguous which of the two candidates is to be chosen. If (l_i, l_{i+1}) was such a line segment with the point p in between then the previous line segment (l_{i-1}, l_i) as well as the next line segment (l_{i+1}, l_{i+2}) do not contain that point while still accounting for the vertices l_i and l_{i+1}. Thus the bearing between p and l can be defined by ignoring this special case:

$$\text{Let } \alpha_{\vec{a}} := bearing(\vec{p}, \vec{a}) \text{ and } \alpha_{\vec{b}} := bearing(\vec{p}, \vec{b}) \text{ and } \delta := \alpha_a - \alpha_b$$

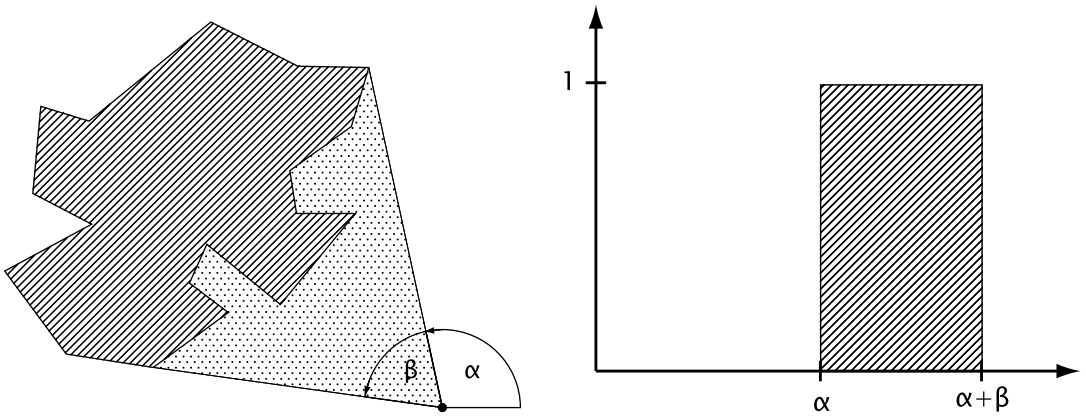
$$bearing(p, l) := \begin{cases} [\alpha_a, \alpha_b] & \text{if } \delta < 0 \\ [\alpha_b, \alpha_a] & \text{if } \delta > 0 \\ [\alpha_a, \alpha_a] & \text{if } \delta = 0 \text{ and } \neg Between(\vec{p}, \vec{a}, \vec{b}) \\ \emptyset & \text{otherwise} \end{cases}$$

Now the bearing from point p to the sequence of line segments L can be defined as well:

$$bearing(p, L) := \bigcup_{l \in L} bearing(p, l)$$

Now as the bearing from p to L has been defined, it may be of interest to consider the definition of a bearing from L to p. This may be described as an interval which contains the bearings of all points lying on the line segments or (in the case of a region) of all points in that region to the point p. The same way an interval $[a, b]$ is limited by a and b which are represented by at least(!) one vertex each, these vertices also account for the boundary points of the interval representing the bearing from L to p. The only difference between the bearing from p to point a and the reverse lies in the inverted angle. Also the vertex (or vertices) that formed the upper margin of the interval now form the lower margin and and therefore the lower now forms the upper.

$$\text{Let } [a, b] = bearing(p, L)$$
$$bearing(L, p) := [b + \pi, a + \pi]$$



(a) Illustration of the bearing (b) Interval representing the bearing

Figure 2.8: The bearing of a point to a polygon.

Chapter 3

Implementation

In this chapter an overview over the GeTS framework as well as some basic definitions are given. GeTS forms the basis of MPLL in the sense, that it serves as a skeletal frame upon which the implementation of MPLL relies. In addition their implementation is described without going into too technical details. For that reason, appendix A may be consulted by the reader interested in those aspects. It provides a walk through demonstrating the implementation of an example MPLL type.

3.1 From GeTS to MPLL

As already mentioned, parts of the *GeTS framework* had been reused to form the skeletal foundation of *MPLL*. It is a very basic, functional language that supports overloading of *custom MPLL functions*. Most basic types like `Integer`, `Float`, `String`, and `Interval` are already implemented. Another not so trivial type is `CompoundType`. It represents the composition of one ore more types serving as parameters and a result type. These again may consist of *compound types* themselves. Also there were some mathematical operations like addition, subtraction, multiplication, and division. An implicit *type conversion* between the various number types exists. This conversion may occur, for example, when adding `Integers` and `Floats` or `Integers` and `Time`.

Furthermore, there are a few predeclared enumeration types such as `true` and `false` which are valid values for a `Bool`. It is the notion to further introduce directional enumerations for MPLL such as `Left` and `Right` or

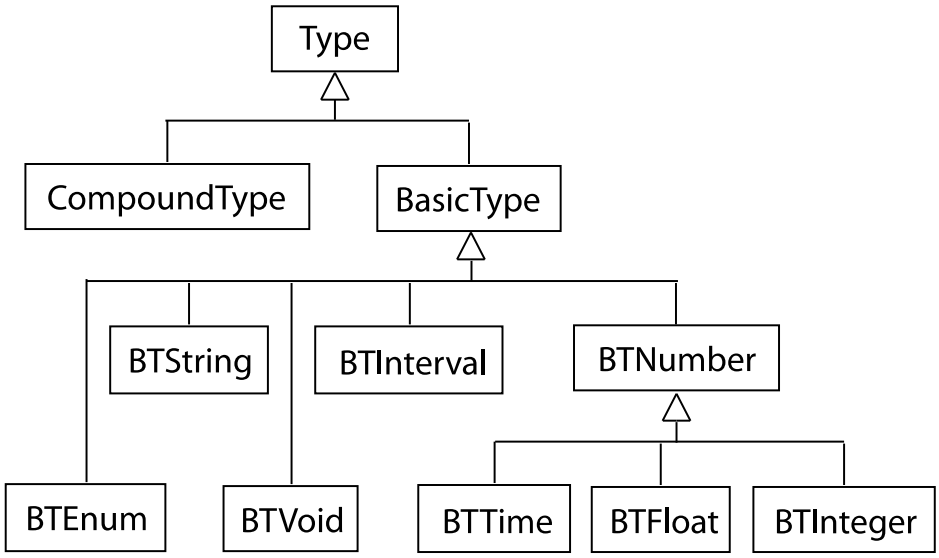


Figure 3.1: A sketch of type classes prior to MPLL.

`North`, `NorthEast`, `East` etc.

The *number types* are summarized in their superclass `BTNumber`. There are many types and corresponding functions. Fig. 3.1 gives a simplified overview of what `MPLL` started off before its actual implementation.

There may be some confusion when talking about *types*. In fact there are a few similar aspects of *type* which derives from the underlying implementation. In appendix A the technical details are elaborated further in detail, however, the following example shall give an idea of the different facets. For one there is usually the C++ implementation of a structure like class. Taking the intended type `Angle` for instance, there is a C++ class that serves as the data structure holding the required values to express an angle in accordance to the *MPLL specification*. This type is then added to the union named `MPLLValue` which serves as a wrapper for all possible *MPLL values*. Thirdly, there is a C++ enumeration called `TypeId` which, as the name already suggests, contains entities that uniquely identify types. There is also a class extending the class `Type` which amongst other possible values and class functions stores said `TypeId`. Lastly, there are several classes belonging to the C++ namespace MPLL. Usually they are either constructors for the `MPLL` types itself or constructors for calls of hard coded MPLL functions. In order to avoid confusion, the following terms are used. The C++ class con-

taining the required values is referred to as either *data structure* or simply its *C++ class*. The entry in the C++ union is referred to as its *MPLL value*, e.g. *MPLL value angle*. The enumeration entry is simply called the *type id* or just *ID* and lastly the C++ class inheriting from `Type` is referred to as the *type class*. Finally, the classes belonging to the namespace `MPLL` with the exception of *type classes* are called *MPLL classes*.

The implementation of MPLL is structured in several layers. Consisting of the scanner and the parser which are responsible for capturing the input, translate it into semantic actions. These actions are handled by the classes and types that are combined in the C++ namespace MPLL and their corresponding type classes.

3.1.1 Scanner

The Scanner is described by the file `scanner.l` and contains the information for the lexical analysis and thus forms the first step of the translation of MPLL statements. Regular expressions and fixed string values are used to preprocess the incoming character stream into an outgoing stream of so called tokens. This token stream is then passed on to the parser.

3.1.2 Parser

On the side of the parser the input is compared and matched to grammar rules. The associated actions then handle the expressions accordingly. There are several stacks which are used for pushing back types and values for further processing. Due to the recursive nature of a given expression the actual value of an MPLL function call can only be determined after the whole expression has been processed. Therefore parsing is done in two steps. First, only types are identified and pushed back into a stack. Second, the actual execution and generation of function instances is done. The separation of these two steps is essential as not all MPLL Types necessarily hold a value yet. The obvious example are MPLL functions itself that are of type `CompoundType` which represents the function's signature. It is intelligible that function definitions may rely on other functions and thus can only produce values if these are executed in a function call. However, this is not the case when their definition is parsed. For future reference the duration of execution of the first step is called “parse time” while the second is called “run time”.

3.1.3 Namespace MPLL

The files `MPLL.h` and `MPLL.cpp` define the C++ namespace `MPLL`. It contains several type definitions and classes. Mostly there are classes corresponding to each `Type` there is. Commonly these are direct or indirect subclasses of `Application`. It represents an entity such as an MPLL Function, an MPLL data type or even syntactic MPLL constructs, e.g if-then-else statements. The parameters of constructors usually include references to `Types` of the input values.

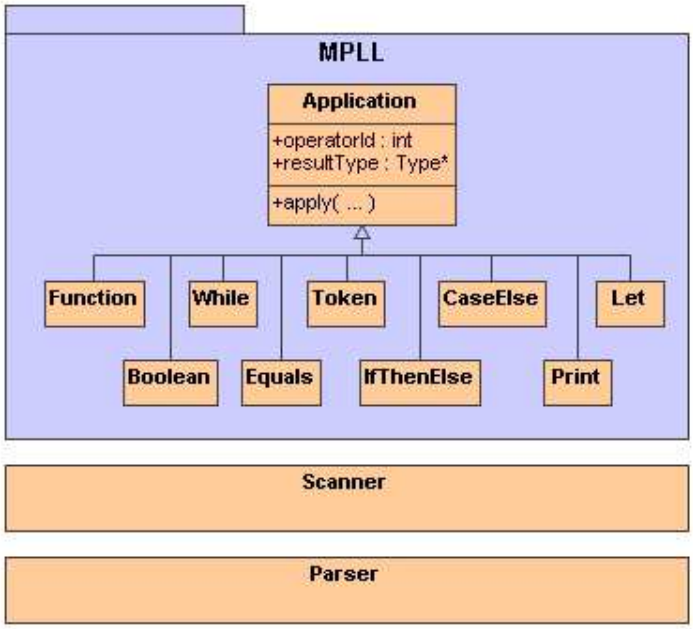


Figure 3.2: Outline of some of the classes inheriting from `Application`

3.1.4 Class Type

MPLL data types are represented by the class `Type` and its subclasses. These belong to the C++ namespace MPLL. They are defined by the files `Type.h` and `Type.cpp`. Among them there are *basic types* and *compound types*. Latter serves as a function's signature. It consists of a return value as well as one or more input types. These in turn are of type `Type`. Basic types are subdivided into several types like `Float`, `Void`, and `Interval` whereas the number

types are subclasses of `NumberType`. Fig. 3.2 shows the interrelation between these types. In addition they inherit the function `apply` which realizes the second parsing step as discussed before. For instance the class `MPLL::Angle` requires three type parameters. After executing `apply`, the actual values are used to instantiate the angle. Likewise more complex constructs can be executed such as `if-then-else` or `let-in` statements in MPLL.

3.2 MPLL

The first mile stone was to implement the additional data types of `Angle`, `Point`, and `Configuration`. At this point a few things can be noted. Firstly, a configuration combines the characteristics of an angle as well as a point. Furthermore it has been previously mentioned that there is an implicit *type conversion* between number types. The code that achieves this *type casting* utilizes a huge number of *if-then-else* and *switch-case* statements. From a certain point of view it can be argued that `Integers` are a subclass of `Floats`. This implicates a *type hierarchy* which is an idea that is elaborated in the following subsections.

Another design decision was to provide only one hard coded function that serves as a constructor for each MPLL type. The remaining functions can be categorized in the sense that they can be associated with one of these types. These functions are assigned to one class for each type and its value and further distinguished by their OP-code to avoid an unnecessary and obfuscating overhead in code. Hard coded functions cannot be overloaded as their string presentation are intercepted by their respective tokens. Therefore the constructors have the prefix "`new_`". For example, `new_angle(Float,Float,Float)` is the said constructor functions of `Angle`. An additional *custom function* `angle(Float, Float, Float)` has been added as a beautifier. This makes it possible to overload the construct `angle`.

3.2.1 Angle

The internal presentation of an angle had been chosen to represent the angle in *Grad*. In order to store the relatively high values without the loss of accuracy, due to rounding errors or simply due to the very limited domain of most types, a `long int` is used for that purpose where the first six digits represent the digits positioned after the decimal point. Hence $90G$ are stored as `90.000.000`.

The first step was to implement a C++ class `Angle.h` / `Angle.cpp` which was intentionally kept very plain. Its mere function was to serve as a record for the minimum and maximum value as well as the value describing the actual angle. According to the attributes a constructor and appropriate get- and set methods were implemented. Unless future MPLL specifications require it, there is no need to add further functionality. Otherwise, it is intended to

be realized in the MPLL standard library.

In order to integrate the class `Angle` as an MPLL type, the class `BasicType` had to be extended as well as some other adjustments to the classes of the namespace MPLL. Appendix A gives a detailed summary of these steps as well as check list that may serve as a guide for future extensions to MPLL.

After having established the MPLL type `Angle` there still was a lack of constructors and functions that enable its creation or the possibility of operating on this type. Thus, the next logical step was to modify the scanner and parser files. A new token had been added to the scanner mapping the string `Angle` to the token named `MPLL_ANGLE`. At this point the decision was made to use the prefix `MPLL_` for all tokens that represent keywords for MPLL constructors or functions. This aims at summarizing and improving the readability of the respective code. The parser had been extended by one syntactic definition of `expression` using `MPLL_ANGLE` to identify the constructor call. The action associated with that syntactic expression invokes the constructor `MPLL::Angle(int,int,int)` that had been added as well. A detailed explanation of the underlying steps can be found in appendix A as previously mentioned.

According to the MPLL specification there are several functions operating on the MPLL type `Angle`. Among them there are `min` and `max`, which extract said minimum and maximum value of the range the value of the angle may be stored in. Further keywords denoting MPLL functions had to be overloaded. For example, `max` is a mathematical function which determines the maximum of two number values. Now it is also used to determine the upper bound of the interval a given angle may be expressed with. In the GeTS framework every keyword is mapped to a single function in C++. It would be undesirable to handle the same keywords that denote semantically different MPLL functions in one function. That is why the action in the parser file first distinguishes both MPLL functions, by comparing the types of the input, before delegating them to the appropriate function.

Furthermore, possible obfuscation of the code is countered by subsuming all related functions under one class in the MPLL namespace. That means that the instantiation of an MPLL function like `max`, `min` or `grad` are summarized in class `MPLL::Angle_Predicate`.

3.2.2 Point

The introduction of points to MPLL was basically done in the same fashion as it was done with angles. First a data structure `Point` was created which stores the two required `Angle` objects representing both coordinates. Like for `Angle` analogue changes and additions were made to the Scanner, Parser, Type class and classes in the namespace MPLL.

3.2.3 Configuration

Configuration was implemented in the same way as it had been done for `Angle` and `Point`. It has been pointed out that configurations are similar to points. That is why the data structure of `DataType::Configuration` extends the class `DataType::Point`. However, this is not reflected in the language layer of MPLL. Both types require functions to access the `X` and `Y` values, for instance. They were treated as two different types entirely. As a consequence both needed to implement a common set of MPLL functions. This redundancy was resolved, which is elaborated on during the course of the following subsections.

3.2.4 Type Hierarchy

Having established `Points`, `Angles` and `Configurations`, the next aim was to implement the data type `Polygon`. This is where the first interesting issues had to be unraveled. Until this point, MPLL as well as GeTS did not have any data type representing lists, sets or vectors or any other data type to serve as a container for values of arbitrary size. The MPLL function `print` possesses a certain similarity to such a function in the sense, that a, arbitrary number of parameters can be stated in a function call. The decision to add a new data type called `List` was made for several reasons. Firstly, polygons could be represented directly or indirectly by accessing a list of points. Secondly, it provides the possibility to handle a set or list of different entities. It is evident that at a later point certain functions must yield results that contain lists of other types than points. For example, the function that triangulates a given polygon must return a list of triangles or at least a list of polygons representing triangles. An even more obvious example is the data type multiline, like a polygon, contains a list of points.

The aspect of a type hierarchy emerges as a `Polygon` can be perceived as

a subclass or rather subtype of a `List`. More specifically it can be seen as a direct subtype of `Multiline` which again is a direct subtype of `List`. As mentioned previously, this is not the only circumstance that suggests a type hierarchy. Summarized there are:

- `Integer` as a subtype of `Float`
- `Point` as a subtypes of `Configuration`. A configuration appears to be a specialized point rather than a specialized `Angle`
- a polygon is a multiline with the additional condition of being closed.
- a multiline is a list which contains points

Possible additional data types are triangles that are merely polygons consisting of three points, or line segments which are multilines containing exactly two points. Alternatively a line segment can be circumscribed as a pair containing points which in turn implicates a `pair` to be a special instance of a `List` of the size two. These have not been implemented in MPLL yet but are mentioned in 8.

Apart from these new possibilities, it is clear that the introduction of a *type hierarchy* adds to the modularity of the preexisting types in MPLL, as well as the ones to be introduced in the future. The considerable amount of different data types in respect to the cumbersome hard coded verification and comparison of types, has a negative impact on the readability of the code and, thus, makes it prone to flaws and errors. Therefore, there is an additional advantage to the hierarchy suggested. Instead of the manual comparison of each type, this process can be simplified by providing the means of an automatic type comparison utilizing a given taxonomy of types. Furthermore, this allows the use of implicit and static type casts. As an additional benefit, most of the preexisting code can be reduced by a notable number of lines of code, namely the ones responsible for the implicit type casts and arithmetic operations. Last but not least, inheriting types do not need to implement the same functions again which results in a reduction of redundancies of line of code in the intended MPLL standard library.

The only issue left at this point, was to decide what type the root of this tree-like order is supposed to represent. It is evident that none of the discussed types can be justified to take its place. Thus, an "artificial" data type had to be introduced. Object oriented languages like *Java* or *Ruby* for example, call

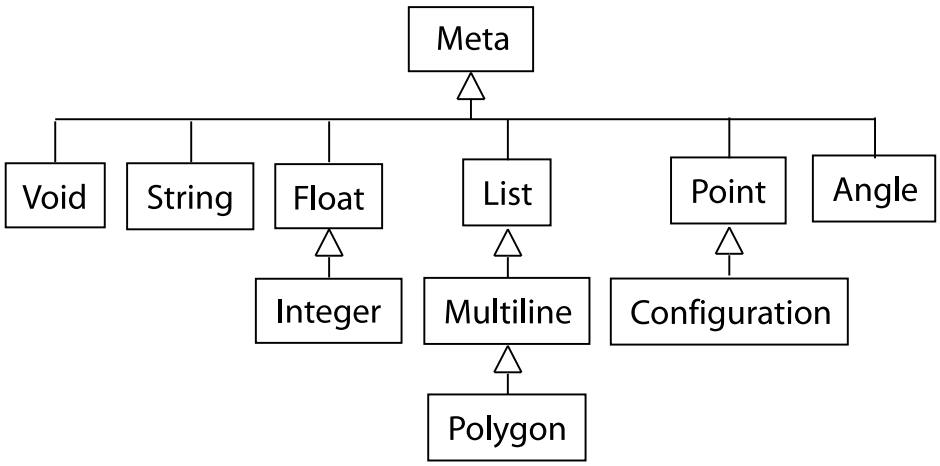


Figure 3.3: Outline of the type hierarchy.

the ultimate ancestor of all other classes "`Object`". As MPLL is not object oriented but functional this name would bestow an odd meaning. Eventually the type in question had been named "`Meta`" as it is a prefix commonly used to indicate abstraction. Figure 3.3 shows the sketch of the original idea of the type hierarchy.

In order to implement such a hierarchy a few steps had to be taken. Firstly, there is a one-to-many relationship between a parent type and its subtypes. Because of that an attribute named `parentType` has been added to the class `Type` which holds a reference to another `Type`. The constructor of every type had been extended by an additional parameter to specify the parent type.

There is a function in the C++ namespace MPLL called `initialize()`. It is responsible for instantiating all types and adding them to a map which holds references to all types. According to the changes every type is initialized with this additional parameter. Due to the fact that the reference to a parent type needs to have been added to the map before it can be specified to be a parent, the order of the initialized types is critical. As a result of this `Meta` needs to be the first type to be initialized followed by its direct subtypes, which again are followed by their direct subtypes etc.

Finally additional C++ functions were added, whose purpose lie in comparing two types in respect to their compatibility. The function `static bool isCompatible(const Type*, const Type*)` for instance, checks whether a

is-a-relationship between the arguments exists. The algorithm is fairly simple and is described as follows in pseudo code. It searches for a match to the expected type by following the tree up to the root until both types match.

```
isCompatible(currentType, expectedType)
    if currentType == expectedType then return true
    else
        if currentType == Meta return false
        else return isCompatible(currentType.parent, expectedType)
```

The following illustrates two examples of the chain of events for such a function call:

```
isCompatible(Configuration, Float)
isCompatible(Point, Float)
isCompatible(Meta, Float)
false
```

```
isCompatible(Integer, Float)
isCompatible(Float, Float)
true
```

The class `Application` had been extended by the attribute `vector<Type*> expectedTypes`. Furthermore every MPLL type constructor has been modified to store their expected parameter types. When the constructor of such a class is invoked it stores the actual parameter types in its attribute `vector<Type*> ArgumentTypes` as well as its expected type parameters. Then it commences a pairwise comparison between the types stored in both vectors. As opposed to the previous implementation, the types are not only accepted if the actual types are identical to the expected ones, but even if the actual type checks positive for the *is-a-relationship* towards the expected type utilizing the algorithm mentioned above.

As already explained parsing is done in two steps. The first merely handles the types involved. With the latest changes this does not cause any problems concerning the newly introduced type hierarchy. The second step needs to handle the actual values. In order to do so, it does not suffice for the types of

the values to be compatible but they also need to realize a type casting to the expected type. A static cast in C++ is not adequate as the data structures do not necessarily (and usually do not) inherit from the classes that maintain the parent types. For instance the type `Polygon` is intended to be a subtype of `List`. While a polygon is represented by a vector of points, a list is a vector not only containing MPLL values but also their types. Another example is the casting of a polygon to a multiline which is introduced later. While both are of type `vector<DataType::Point*>` they are interpreted differently. A polygon whose vector stores the points p_0, p_1,...,p_n is implicitly assumed to have an additional point at the end that is congruent to the first such that it forms a closed graph. If the vector associated with a multiline stored the same points it would not result in a closed graph as the last point would be missing. Thus a cast from `Polygon` to `Multiline` needs to take this into account by adding the last point explicitly.

For these reasons additional functions were added. Among them there is `MPLLValue castType(Type*, pair<Type*, MPLLValue>)` which casts the value, given by the second argument, until its type matches the first parameter. This is done in an analogue fashion as it has been done with `isCompatible(...)`. The only difference is that it casts the value stepwise to the parent type until it matches the expected type. It is noteworthy that this function is the only one that needs to explicitly check types by using a *switch-case statement* in order to realize the implicit type and value conversion. It makes all such hard coded comparisons obsolete, which previously took up hundreds of lines of codes.

3.2.5 Lists Part I

Implementing the new data type `List` was done in a similar fashion like the function `print`. The parser file contains a recursive definition of `expressionList`.

```
expressionList:
  {}
  | expression  {MPLL_TypeVector.back().push_back($<type>1);}
  | expressionList ',' expression
     {MPLL_TypeVector.back().push_back($<type>3);}
;
```

As displayed, the action pushes each expression onto a stack. This stack is then read and stored in a vector and assigned to the MPLL value `MPLLValue.lList`. The list constructor is an exception to the other constructors. Like `print` it accepts an arbitrary number of parameters. There are no means for custom created MPLL functions do not allow this yet. As a consequence there is no other constructor serving as a beautifier nor as an alternative constructor.

3.2.6 Generics

At this point another issue emerges. Constructing the `List` is rather simple, but accessing its elements is not. In fact every function must have a signature, i.e. the types of the parameters as well the result type have to be known during *parse time*. MPLL is not strictly typed in the sense that it can determine a result type during *run time*. But this only works for direct invocation of MPLL functions. When implementing further *custom MPLL functions* which are based on other *MPLL functions*, no matter whether they are hard coded or not, the result types of the underlying functions have to be known during *parse time*. The following examples illustrates the problem. The function `element(List list, Integer index)` is the function that returns the element of a given list at a certain position.

example 1:

```
List list1 = List(1,2,3,4,5)
```

`element(list1, i)` must return an `Integer` for $i \in \{0, 1, 2, 3, 4\}$.

```
List list2 = List(List(1,2,3), List(4,5,6))
```

`element(list2, i)` must return a `List`. But that is not all the list must return. A call `element(element(list2,1),0)` has to return the `Integer` with the value of `4`. In other words the type `List` has to contain information that allows the retrieval of the correct type of an element. The following example demonstrates why the contained types have to be known a-priori.

```
Integer:checksum(List list) =
   if empty(list) then 0
   else head(list) + checksum(tail(list))
```

This results in an MPLL error as the return type of `head` is unknown. This could be worked around by defining `checksum` in the same context in which `List` had been constructed in. Nevertheless this does not solve the problem but merely delaying it. Furthermore in the recursive call the parameter is `tail(list)` which constructs a new list. At this point the context is lost and the contained elements cannot be determined anymore.

It can be concluded that MPLL up to this point, did not provide any means to circumvent the problem. There are two approaches to solving this problem. One is to allow dynamic type casting the other is to allow *generics* which are known in languages like *C++* or as a recent addition to *Java*.

The type casting approach has several disadvantages. It results in MPLL code being cluttered by cast functions. This approach would also have a strong negative effect on type safety as mpll or GeTS do not provide the possibility to throw and handle exceptions. On the contrary, *generics* add to the readability and type safety. Furthermore they suggest the addition of templates that allow for the introduction of polymorphism to MPLL. This works well with the previous adoption of type hierarchies.

Implementing *generics* in MPLL began by extending `typeExpression` in the parser. Additionally to the keyword denoting a type, an `expression` is followed by recursive declaration of further type expressions. In regard to the notation of C++ and Java the type of a list which, for example, contains integers can be declared as `List<Integer>`. In the same manner a definition like `List<List<Angle>>` is thinkable. On the parser level it is being made sure, that the type described is a *generic type* when using this kind of declaration. On the contrary no *generic type* may be declared without it.

The next step was to add another attribute `genericType` to the class `Type`. Consequently, a list containing integers is the instantiation of the basic type `List` where its `genericType` holds a reference to the basic type `Integer`. Likewise `List<List<Point>>` is represented by the instance of `List` with `genericType List` which again encapsules `Point`. As already mentioned in 3.2.4, a static map of basic types is kept for the sake of easier and more efficient referencing. Unfortunately there is virtually an unlimited number of possible generic type expressions. Naturally it is counterproductive and inefficient to add each declared generic type to the map. Furthermore `BasicType` had been used in a static manner hence. The introduction of generics changes

this to a certain extent. The following solution to this problem assures all former basic types to be treated the same way as it had been done previously. The exception is, that each new instance of a generic type can reference to its own `genericType`. Therefore they are not added to the map. At this point it should be noted that a generic type may only contain one type parameter. In C++, for instance, the use of the class `pair` is quite common and requires two type parameters, i.e. (*) `pair<int, float>`. MPLL does not implement multiple type parameters, because there has been no need for this yet.

With the introduction of the type hierarchy, additional care had to be taken by C++ functions that are responsible for type comparison. Likewise, this is the case with *generics*. It does not suffice anymore to compare `typeIds` and the parents `typeIds` but also to include their underlying types. As a corollary, the C++ function `bool Type::isCompatible(const Type* ty1, const Type* ty2)` had been modified to meet the new requirements. The changes are illustrated by following pseudo code:

```
1 isCompatible(currentType, expectedType)
2    if currentType == expectedType then return true
3    else
4       if currentType.type == expectedType.type then
5          return isCompatible(currentType.genericType,
6          expectedType.genericType)
7       else if currentType == Meta then return false
8            else return isCompatible(currentType.parent, expectedType)
```

The following examples illustrates the workflow of this algorithm.

- Comparison between `List<Float>` and `List<Integer>`:

```
1 isCompatible(List<Integer>, List<Float>)
2 isCompatible(Integer, Float)
3 isCompatible(Float, Float)
4 true
```

- Comparison between `List<Angle>` and `List<List<Angle>>`

```
1 isCompatible(List<Angle>, List<List<Angle>>)
2 isCompatible(Angle, List<Angle>)
```

```
3 isCompatible(Meta, List<Angle>)
4 false
```

- Comparison between `List<Polygon>` and `List<List<Angle>>`

```
1 isCompatible(List<Polygon>, List<List<Angle>>)
2 isCompatible(Polygon, List<Angle>)
3 isCompatible(Multiline, List<Angle>)
4 isCompatible(List<Point>, List<Angle>)
5 isCompatible(Point, Angle)
6 isCompatible(Meta, Angle)
7 false
```

- Comparison between `Polygon` and `List<Point>`

```
1 isCompatible(Polygon, List<Point>)
2 isCompatible(Multiline, List<Point>)
3 isCompatible(List<Point>, List<Point>)
4 isCompatible(Point, Point)
5 true
```

3.2.7 Templates

With the introduction of *generics* it is now possible to extend MPLL to support *templates*. These serve as *type variables* similarly to the polymorphic types in `SML`. Without those polymorphic MPLL functions would not be possible. While the function `isempty(List)` does not require any knowledge of the types contained in the list and merely relies on `size` other functions do. `head`, for instance, is supposed to return the first element of a list and utilizes `element`. In fact the implementation is simple: `head(List<T> list) = element(list, 0)`. Element itself is hard coded but has an explicit signature `List<T>` $\mapsto$ `T`. Therefore it can be determined that head has the same signature. This already insinuates that special care has to be taken when unifying or partially matching the templates when determining their types. For example it could have been stated that the signature of `element` is `List<S>` $\mapsto$ `S` substituting `T` by `S` which must not cause any problems as `S` can be matched to `T`.

In order to implement these new features, the parser has to be altered again. In the former case, the actions that handle type expressions threw an excep-

tion if the given token does not match any keywords denoting types. Now the parser had to be changed to accept such arbitrary, non-matchable tokens and use them as a placeholder for a type. In other words a declaration of `List<T>` could not be recognized as `T` is not a type. With the latest alteration it has become a valid expression. However, `T` needs to be handled by MPLL. For that reason a new type, called `PolyType`, was added extending the C++ class `Type`. Firstly, it stores a string containing the name of the polymorphic type. In the above example it would simply store the string `"T"`. Secondly, it holds a reference to an actual type. Of course this type is not known from the beginning and is resolved at a later stage of execution.

The existence of templates complicates comparison between types further. From there on, it is not only necessary to check the relationship of two types in their hierarchy and, in case of *generics*, their referenced types' compatibility, but also the possible compatibility of *templates* which assume the role of placeholders for types. The introduction of *generics* made it necessary to alter the function `isCompatible(...)`. From now on, *templates* have to be taken into account as well. The new implementation is again given in pseudo code and contains only a minor alteration to the previous one:

```
isCompatible(currentType, expectedType)
  if isPolyType(expectedType) then return true   // new line of code
  if currentType == expectedType then return true
  else
    if currentType.type == expectedType.type then
      return isCompatible(currentType.genericType,
      expectedType.genericType)
    else if currentType == Meta then return false
      else return isCompatible(currentType.parent, expectedType)
```

It can be seen that any type is deemed to be compatible with a template. This expresses the intention that a template is just a placeholder for another type which can be matched to any given type. Nevertheless, this is only half the truth. As soon as a polyType has been matched to a certain type it should be bound to it. This may cause a clash with another type that could be matched to the template as following example shows:

Given a definition of a custom MPLL function.

```
foo(T t1, T t2) = List(t1,t2)
```

There is nothing harmful if this function was called by two parameters of the same type like in `foo(1,2)`. It is clear that the result type should be `List<Integer>`. If the call would involve two different types, like an `Integer` and a `Float`, the template `T` can be matched to both types. It suggests itself that due to the compatibility `T` should be matched to a `Float` and hence the result type ought to be `List<Float>`. This cannot be resolved in case of two types that are not compatible at all such as in the case of `foo(Angle(), Point())`. It is self-evident that this must either result in an error or in a list with a useless type of `List<Meta>`.

After the familiarization of this problem there are several approaches to solving it. It is neither necessary nor reasonable to realize such a type matching algorithm in `isCompatible(Type*, Type*)`. The issue occurs in function declarations. Moreover, there is a difference between a function declaration and a function call. In the latter, actual instantiated values and hence types are available as opposed to the former. But even a declaration bears the same issues. For instance, a function could rely on the definition of one or more other functions. In such a case the signatures must be compatible and the involved templates resolved accordingly.

To demonstrate the implications two functions can be provided:

```
foo2(R r, S s) = s
bar2(T t1, T t2) = foo2(t1, t2)
```

The signature of `foo2` is `R * S` $\mapsto$ `S`. The result part of the signature of `bar2` has to be derived from the former. A pair-wise comparison yields that `T` $\equiv$ `R` and `T` $\equiv$ `S`. Therefore `bar2` must be of type `T * T` $\mapsto$ `T`. Although it may also be expressed as `S * S` $\mapsto$ `S` it implicates a notional mistake. The implemented compatibility checks are transitive, reflexive and antisymmetrical. That means that `X` may be compatible to `Y` but not necessarily `Y` to `X`. Moreover a type can only be matched to itself or a more general type. The following exemplifies why the opposite is not possible:

```
foo3(T t1, T t2) = List(t1,t2)
bar3(R r, S s) = foo3(r, s)
```

The signature of `foo3` is `T * T` $\mapsto$ `List<T>`. When resolving the result type of `bar3`, `T` is mapped to `R` and then to `S` which causes a clash. Of course, it might be possible that `R` and `S` are indeed of the same type, but

it is not necessarily the case. This becomes clear when using concrete types instead. For example, `new_bar(Integer i, Point p) = foo3(i, p)` definitely causes an unresolvable situation. To be more specific, it shows that only the types of the signature of the underlying function definitions may be mapped to concrete parameter types. The opposite is to be avoided. As an unintended consequence this avoids possible problems of clashing `PolyType` names as following example demonstrates:

```
foo4(R r, T t) = r
bar4(T t) = foo(t, 3)
```

The signature of `foo4` is `R * T` $\mapsto$ `R`. Both functions, `foo4` and `bar4` contain templates by the name `T`. To better distinguish the domain of the involved templates they are labled by the names of their respective function in subscript. It is clear that these do not stand for the same types. When determining the signature of bar it becomes evident that $\texttt{R}_{foo4}$ is mapped to $\texttt{T}_{bar4}$ and $\texttt{T}_{foo4}$ to the type `Integer`, as implicated by `3`. The circumstance, that only templates from the underlying function `foo4` are mapped to templates of the calling function `bar4`, makes it unnecessary to introduce means of artificially distinguishing templates by the same name.

Following this examination an algorithm that realizes type matching can be provided. It is applied to the function `FctInstance::FctInstance(string* name, const vector<Type*>& types, Function* FCT)` which handles instances of MPLL functions. Again this algorithm is given in pseudo code followed by an elaboration of its steps:

```
1  Type getResultType(Function f, Types[] actualTypes)
2      Types[] expectedTypes = f.getArgumentTypes()
3      MAP<string,type> mapping
4      FOR int i = 0 TO expectedTypes.length
5         Type aType = actualTypes[i]
6         Type eType = expectedTypes[i]
7         IF containsTemplate(eType) THEN
8            Type match = MATCH_TEMPLATE_TO_TYPE(eType, aType)
9            PolyType poly = EXTRACT_TEMPLATE(eType)
10           IF (mapping.contains(poly.name) THEN
11              Type oldMatch = mapping.get(poly.name)
12              IF isCompatible(match, oldMatch) THEN DO_NOTHING
13              ELSE IF isCompatible(oldMatch, match) THEN
14                 mapping.replace(poly.name, match)
```

```
15              ELSE ERROR
16           ELSE mapping.add(poly.name, match)
17        ELSE IF containsTemplate(aType) THEN ERROR
18     Type resultType = f.resultType
19     IF containsTemplate(resultType) THEN
20        Type template = EXTRACT_TEMPLATE(resultType)
21        Type match = mapping.get(template)
22        resultType.replace(template, match)
23     RETURN resultType
```

This algorithm maintains a map that associates a template name with a type (line 2). It iterates over every argument type (line 3). If the current argument type contains or even is a template type (line 6) the corresponding actual `Type` is searched in order to find the pendant to the template (line 7). If it is the first occurrence of this template the matching type is added to the map (line 15). Otherwise it is already stored in the map. The intention is to only map the most general occurrence of said matches. Therefore the new match and the already stored match for the same template are compared by the use of `isCompatible`. By doing so, the more generic match is determined and if necessary replaces the old one (lines 10 to 14). If both of them are incompatible an error occurs (line 14). The same happens if the actual `Type` contains a template while the expected type does not (line 16). Finally the result type of the function in question is extracted. In case it consists of a template, it is substituted by the appropriate match which is among the previously determined mappings.

A few examples follow that illustrate the execution of this algorithm.

```
addition(Float f1, Float f2) = f1 + f2
badIncrement(T t) = addition(1, t)
```

The `+` operator is only defined for number types. But the reason why MPLL declines the last function, is because `Float` cannot be made to fit template `T`. Line 15 applies.

```
identity(T t) = t
foo(Integer i) = identity(i)
bar(S s) = identity(s)
```

the declaration of `foo` as well as `bar` are valid. In the first case `T` is the expected type for `identity`. The actual Type is an `Integer`. Thus, `T` is

mapped to the Type `Integer`. Therefore `foo` is of type `Integer` $\mapsto$ `Integer`. In the case of `bar` the same happens. But this time `T` is not mapped to a basic type, but mapped to another template `S`. Thus, the signature is `S` $\mapsto$ `S`.

```
head(List<T> list) = element(list, 0)
```

As it is later shown, `element` is a hard coded function with the signature `List<T>` $\mapsto$ `T`. The function `head` is part of the MPLL standard library as illustrated in appendix B. The expected type for function `element` is therefore `List<T>`. The algorithm determines $\texttt{T}_{head}$ to be a match for $\texttt{T}_{element}$. Before the result type is returned its occurrence of `T` is substituted by its homonymous counter part `T`. This makes sense as both templates only happen to share the same name by coincidence.

```
polyFoo(Polygon p) = head(p)
```

This example is more complex as it involves implicit type conversion. In line 7 the function `MATCH_TEMPLATE_TO_TYPE` tries to determine the match between the expected type `List<T>` and the actual type `Polygon`. The sequence of actions can be illustrated as follows.

```
1 match (Polygon, List<T>)
2 match (Multiline, List<T>)
3 match (List<Point>, List<T>)
4 match (Point, T)
5 return [T -> Point]
```

The result is that template `T` is mapped to Point and thus the signature of `polyFoo` is resolved to `Polygon` $\mapsto$ `Point`.

```
first( R r1, R r2 ) = r1
z1() = first( Angle() , Point() )
z2() = first( Configuration(),Point() )
z3( Polygon p, Multiline m, List<Point> l ) =
   first( first( l , m ),p )
```

The signature of `first` is `R * R` $\mapsto$ `R`. The declaration of function `z1` fails. `R` is mapped to type `Angle` first. Then another match of type `Point` is found but both matches are incompatible (line 14). In a similar manner `R` is mapped to `Configuration` in function `z2`. However, the declaration does

not fail as the other match for `R` is of type `Point` and as such is compatible to `Configuration`. The mapping is updated by assigning `R` to `Point`. Because of that the signature is `Configuration * Point` $\mapsto$ `Point`. In order to determine the signature of z3, the nested call to function `first` needs to be resolved. It is `List<Point> * Multiline` $\mapsto$ `List<Point>`. Therefore the signature of the wrapping call to function `first` is determined to be `List<Point> * Polygon` $\mapsto$ `List<Point>`.

The previous subsections used Lists, Multilines and Polygons as examples of types to demonstrate *type hierarchy*, *generics* and *templates* whilst anticipating their actual introduction. The following subsection makes up for that.

3.2.8 Lists Part II

`List` has already been halfway established. A feature that had been incorporated, is that the actual type of a list is determined during its construction. When the provided parameters are of mixed type the resulting type should contain the most general type of the parameters. Given a list by `List(Point(),Configuration())` for instance results in a `List<Point>`. Another example is `List(Polygon, List<Point>, Multiline)` where the least specific type is a `List<Point>` and as such the resulting type needs to be `List<List<Point>>`. However, a List defined by an `Angle` and `Point` has only `Meta` as a common denominator and such should return a `List<Meta>` which, admittedly, is of almost no use.The following algorithm is similar to a search for a maximum. The `isCompatible` predicate is used as a comparator. Nevertheless, due to the fact that the type hierarchy is a non-linear structure the described order cannot be connex. In other words, two arbitrary types do not necessarily stand in the *is-a-relationship* but could be neighbours or twins. In that case they have a direct or possible indirect parent in common.

```
1 Type getCommonType(Type type1, Type type2)
2    IF isCompatible(type1, type2) return type2
3    ELSE IF isCompatible(type2, type1) return type1
4    ELSE return getCommonType(type1.parent, type2)
```

To illustrate the process the following example can be given:

```
1 getCommonType(Polygon, List<Configuration>)
2 getCommonType(Multiline, List<Configuration>)
3 getCommonType(List<Point>, List<Configuration>)
4 return List<Point>
```

This algorithm is not only used for constructing new lists but also appending them. A hard coded function, for example, is `append [List * List` $\mapsto$ `List]`. The type that serves as a common denominator has to be found, using the types of both arguments and apply said algorithm.

There are four hard coded MPLL functions. These are `size`, `element`, `tail` and `append` which are all implemented in the class `ListPredicate`. Any other required function can be realized by a custom MPLL function as shown in appendix B. Casually spoken, these four functions could be described as the analogy to a functional complete system of all required MPLL list functions.

- `size`
  As the name already suggests it returns the number of elements of a list in the form of an integer. As MPLL values store lists as vectors of their values and types, implementation of this function was more then trivial.

- `element`
  The function `element` was actually the first function that led to considering the introduction of generics. A default return type `Meta` would have sufficed as a workaround, but implementation of any further MPLL function utilizing `element` would be futile. Once again, the implementation was trivial as the element of said `vector` at the specified index of the needs to be returned. This function required hard coding because there must be means to access single elements of a list upon which other custom functions must rely on.

- `tail`
  Like most function languages head and tail are common functions to access functions. Whilst head can be realized by `element`, `tail` poses the possibility to create sublists of any given `List` and as such is necessary to be hard coded.

- `append`
  Two compatible `Lists` can be concatenated to form a new instance of `List`. It is the only mean to create a bigger list than one already given. That is why it needs hard coding. From that point of view `tail` and `append` are pendants. One provides the means to create shorter lists while the other enables the construction of bigger ones.

3.2.9 Multiline

Multilines are direct subtypes of `List<Point>`. As a consequence they *inherit* all list functions. In accordance to (?) a multiline stores $n \in \mathcal{N} \backslash \{0, 1\}$ points which define the boundaries of $n - 1$ and thus at least one line segments. Currently there are no MPLL specifications concerning `Multilines` nor were any additional functions added yet. As there are several algorithms that may be applied to polygons as well as multilines their introduction may be a useful future addition.

3.2.10 Polygon

It had been decided to make `Polygon` an indirect subtype of `List`. For the sake of completeness `Multilines` had been added but only after the implementation of `Polygon` and were later integrated to become a direct supertype of `Polygon`. The only hard coded function so far is `triangulate`. The algorithm described in 2.1.6.4 was implemented using two scan sweeps. It is too complex to be realized in MPLL itself nor would it be any efficient.

Further functions were implemented as custom MPLL functions such as `area` and `com`. The former utilizes `triangulate` to add up the areas of the triangles stored in the returned list. The latter is an abbreviation for “center of mass” and works in a similar fashion. Both can be looked up in appendix B.

3.3 Testing and Running MPLL

Until now multiple aspects and their implementation in MPLL have been discussed. One issue has not been addressed yet: getting MPLL statements to execute. For the most time during the work on MPLL a simple shell script was used. It invokes the main method of the class `MPLLTest` providing a few arguments for each line of MPLL code.

This class is a remnant of GeTS. In addition to the pure GeTS/MPLL syntax a few abbreviated syntactical expression were allowed. They are useful for testing purposes. All previous basic types and other classes extending `Type` included functions called `parse(...)`. These functions serve the purpose of parsing these expressions in order to provide an instance of their type on basis of that expression. For example, the instance of a value of type `Time` is derived from the string `"5T"`. It can be argued that this leads to certain inconsistencies as the syntax is nonuniform. Furthermore the `parse` function of `Interval` parses an integer. The arguments provided after this one in the shell script express instances of `Interval`. The integer value is then used as an index in the way that the interval at the specified position is taken as the resulting instance.

Apart of these inconsistencies the script has further short comings. For every line a new instance of the MPLL environment is initiated and right after its execution terminated. This makes it impossible to keep previous function declarations and therefore rules out any means to save a library of *custom MPLL functions*. In addition there were no means of persistently storing nor loading said library in and from a separate file.

3.3.1 MPLLVM

In order to counter the lack of previously mentioned short comings, an additional class had been created to add desired functionality. It utilizes several preexisting parts and unifies them under a common denominator. It also makes use of function constructors in order to create new MPLL function instances, as well as add them to the MPLL namespace. For instance, the constructor initialized the MPLL types as it had been done by running the script. Furthermore, it does not rely purely on the discussed MPLL syntax and refrains from using the mentioned abbreviated notation. In addition two file formats were devised. One fills out the role of storing MPLL function

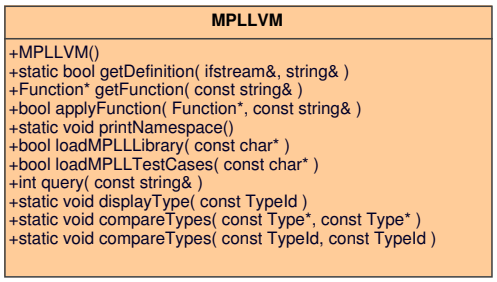


Figure 3.4: UML diagram of the class MPLLVM

declarations for the purpose of representing as MPLL standard library; the other serves as a format for describing test cases for function calls.

Fig. 3.3.1 illustrates the class MPLLVM. There are a few functions that are noteworthy:

- `Function* getFunction(const string& definition)` contains one string parameter which is supposed to represent an MPLL function declaration. It delegates the parsing and function creation by calling the constructor of the class `Function`. Therefore `getFunction(...)` serves as a convenience method. Additionally every successfully created function is automatically added to the current MPLL namespace and returned.

- `bool MPLLVM::applyFunction(Function* fct, const string& params)` ,on the other hand, applies the string representation of MPLL function parameters to a given instance of a `Function`. The result of the function call is then displayed. In the event of a successful execution the function returns `true`.

- In the `main()` method a new MPLL namespace by the name of `"mpll"` is created. It is then set to be the root namespace as well as the current namespace. Therefore every created Function is automatically added to this specific namespace.

3.3.2 MPLL Standard Library file format

The former sections mentioned a so-called *MPLL standard Library*. Most programming languages or frameworks provide APIs or default libraries. The

idea of the MPLL standard library is to provide a minimalistic set of hard coded MPLL functions. These form a foundation enabling the possibility to add further custom MPLL functions which fulfill the MPLL specification. For that reason a file format had been devised. It stores a set of custom MPLL function declarations. These definitions need to be separated by some means. The apostrophe is never used in MPLL expressions. Because of that, every declaration of a function is wrapped at the beginning as well at the end by this character, in a similar manner in which strings are represented.

The convenience method that provides the possibility to read a single function declaration is called `bool MPLLVM::getDefinition(ifstream& in, string& result)`. It uses a file stream to read character by character. As soon as the first apostrophe is read it starts to store the following characters in the string named `result`. After the second occurrence of an apostrophe it exits the function. The boolean value `true` is returned in order to indicate its successful execution.

The function `bool MPLLVM::loadMPLLLibrary(const char* fileName)` puts everything together. It uses `getDefinition(...)` to extract the string representation of each stored MPLL function definition. Then the actual functions are created by utilizing `getFunction(...)`. In case of a malformed file content its execution is stopped. Otherwise all declaration were loaded and stored in the MPLL namespace called `mpll`. From now on it is possible to define functions relying on previous definitions as these are looked up implicitly. The file extension for the implemented library has been chosen to be `"*.mpll"` thus naming the file `stdlib.mpll`.

3.3.3 Queries

Although the means to provide an MPLL library is in place it is not possible to make use of it nor execute single statements. It is possible to add a new function containing such a statement but it would also be stored in the MPLL namespace. For instance the creation of a new instance of `Angle` could be done by declaring the function `a = AngleGrd(45.0)`. If the expression is accepted it displays its type and value, then adds it to the namespace. Persistent storage in a namespace is not desirable for such declarations for they are most likely used temporarily. For that purpose a workaround had been devised. The idea is to allow the *body* of a function to be specified, e.g. `AngleGrd(45.0)`, and then provide a dummy function name which is append

in front of it. After the function has successfully been parsed, its value and type determined and then added to the MPLL namespace. Afterwards it is deleted again. This has been implemented in `int query(const string&)`. The provided string is appended to `"query = "`. In the above example that would result in `"query = AngleGrd(45.0)"`. In case of an error caused by a malformed definition it returns the integer value `-1`. Depending on the successful application of the function a value of `1` or `0` is returned. After that the line `Namespace::deleteAll("query");` is executed, which erases all functions by the name of `query` from the current namespace. As a consequence no MPLL function belonging to the standard library should bear this name. In spite of that it does not pose any serious restrictions on MPLL. At this point it may be noted that alternative and abbreviated syntax fulfilled exactly this purpose. As a result it has been made obsolete.

3.3.4 MPLL Test file format

The function `query(...)` provides a useful way of testing single MPLL statements. It is desirable to store these test cases. Cluttering C++ code with such hard coded function calls is to be avoided. It not only reduces readability of the code but also requires the class `MPLLVM` to be recompiled after every changed or added test. Because of the circumstance that there is already a file format in place that realizes reading function declarations, it is obvious that it can be used to read said test cases.

In analogy to `loadMPLLLibrary(...)` the function `loadMPLLTestCases(...)` has been implemented. Again it uses `getDefinition(...)` to extract strings wrapped by apostrophes. As a small addition this file format, it not only stores the actual queries, but is followed by another wrapped string containing a `+` or `-` symbol. The intention is to express whether the statement is to be executed successfully or not. As an example it could look like this:

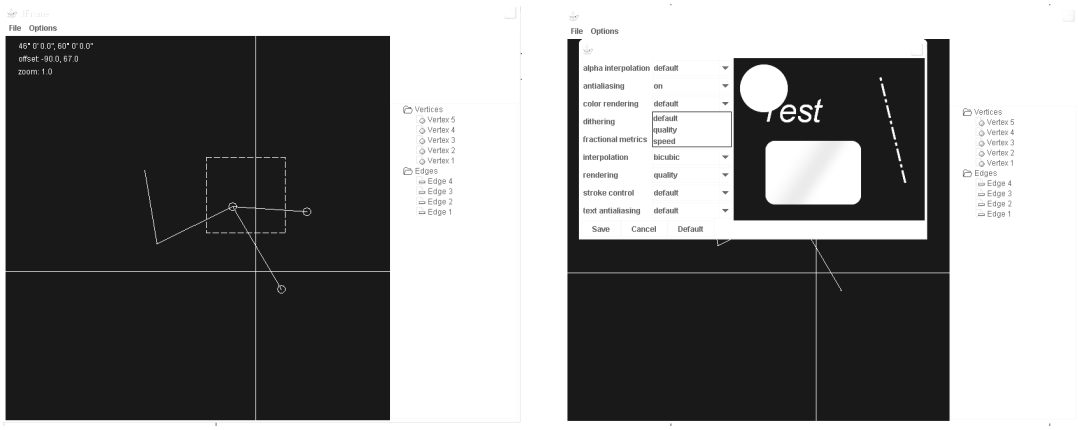
```
'Angle(1.3,2.0,3.0)' '+'
'Angle(1.3,3.0,2.0)' '-'
...
'newList(1.0,2)' '+'
'newList(1,Angle(1,2,3))' '-'
```

The function `loadMPLLTestCases(...)` reads the first string and obtains the test query. It then reads the next string containing the minus or plus

symbol. The first string is then used as a argument of the function call to `query(...)`. In case of a malformed content it stops execution. Otherwise it displays the successful or unsuccessful execution for each test case. This is done by comparing the expected outcome with the actual result from `query(...)`.

3.4 Java Client

Parallel to MPLL a small Java application had been implemented. It is an editor for setting, moving and deleting points. Creation of edges and connection of points is also possible. Furthermore they can be selected and deselected. This application had been created with an MPLL client in mind. Future, implementations may incorporate a console and the possibility to connect to MPLLVM. The display can be used to display results or provide a visualize queries. The implementation is very basic. It wraps user input in an own class in order to provide a common way of handling different types of events. In addition rendering hints such as anti-aliasing and other performance options can be customized.



(a) Points being selected by a mouse drag

(b) Rendering options

Figure 3.5: Screenshot of the Java application

3.4.1 Custom Java API

Even though not directly related to MPLL a Java package is being provided with this work. It contains a number of classes that represent common geometric entities such as angles, points, polygons, and different kind of lines. All of the discussed algorithms are already implemented. The Java client had been implemented separately from this package. In spite of that an easy adaption is possible.

Chapter 4

Conclusion

While GeTS served as a foundation for MPLL, the framework has undergone some major changes. A new feature is the type hierarchy which contributes not only to a cleaner structure, but also eases the maintenance of already existing types as well as future types. Also implemented was the type `List`, a useful tool for handling a set of values. In addition the introduction of *generics* provides not only type safety, but better readability of MPLL code. With the introduction of MPLLVM an application has been established, that enables the maintenance of a standard library as well as a convenient test suite. The introduction of sophisticated language constructs allows to divert the focus on more advanced aspects.

4.1 Further work and outlook

The following points elaborate on the things that are left to do, as well as suggest a few ideas for future extension of mpll.

1. Despite the advancements of MPLL there are still a few issues that need to be resolved before any further work is advised. During this thesis it has been pointed out that a significant amount of code has been rendered obsolete. Due to the lack of time it was not possible to thoroughly clean up the legacy code. Especially the parts of MPLL that are responsible for handling types may still contain bugs. Moreover, several parts need to be refactored to meet the new standards.

2. An aspect of the framework that can be noticed during this work,

is that it needs to be further modularized. Currently there are two types of MPLL functions: custom ones and hard coded ones. The hard coded ones are firmly integrated in the framework. The same counts for MPLL types. It is desirable to out source these parts. MPLL could be made fully customizable concerning types and functions or even their libraries. It is imaginable that external files could store a description of the type hierarchy or its extensions. These extension could be easily stored in a more contemporary format like `XML`. The extension is then applied dynamically to MPLL. Concerning the hard coded functions, they will still need to be hard coded. However this could be done in external modules or classes and dynamically integrated as well. For instance, all former hard coded MPLL functions could be invoked by custom ones. As a fictive example this could be illustrated like this:

```
load("mpll\angle\angle.type")
Angle(Float f1, Float f2, Float f3) =
   call("mpll\angle\angle", f1, f2, f3)
```

The imaginary function `load` retrieves the type definition of angle and appends it to the type hierarchy. The function `call` is the only hard coded MPLL function. It is responsible to invoke external implementations of MPLL functions. Its first argument accepts the string representation of the relative path to a *module*. All further arguments are passed onto the module as parameters of the function call. Furthermore access to non-local systems could be realized in the same manner.

```
load("mpll\graph\node.type")
load("mpll\graph\edge.type")
load("mpll\graph\graph.type")
shortestPath(Graph g, Node start, Node end) =
   call("212.202.192.156:30",
        "TransRoute\shortestPath", g, start, end)
```

This illustrates the idea that MPLL could also query other specialized systems such as *TransRoute*[5], thus delegating function calls. Both examples demonstrate the idea to out-source most functionality. The obvious advantage is a more concise MPLL. This could avoid additional

overhead that will be caused by the introduction of further functions and types. Currently the MPLL types and their functions are "too" integrated. It should rather provide abstract means of future extensions than serve as compound of an arbitrary number of types and operators. The requirement on the *modules* would be simple. They need to implement a certain interface. It should return the result type as well as its value for the function in question. Furthermore, the core application is not required to compile after a change had been made, a factor that makes working with MPLL cumbersome.

As an additional benefit, the introduction of modules can also provide support for other programming languages. It can be argued that C++ is not a contemporary language anymore. Programs written in this language can be quite efficient, but at the same time they are more prone to possible bugs. They require an overhead in effort while at the same time they do not provide any means of producing more readable nor safer or agile code. Therefore a transition to more contemporary languages might be considered. Even though there is no immediate need to do so, modules could already be implemented in other languages. For instance, an interface definition language (IDL) like `CORBA` could be used to communicate between the main application of MPLL and *modules* written in different languages. In that way a step-wise transition from C++ to another language may be realized without compromising the whole framework.

3. Another thought can be given to databases. GIS data commonly requires a huge amount of space. It is impractical to tunnel these enormous amounts of data from one system to the other by the means of MPLL types. Instead a way of referencing the data needs to be devised.

4. It is possible to store functions in a MPLL namespace. So far these serve as a library. Most languages allow to declare and instantiate variables. Variables storing generic values are nothing else then 0-ary functions. Therefore MPLL possesses the ability to store arbitrary variables. In order to avoid mixing the standard library with temporary or session depending variables future extensions could store these under a different namespace, e.g. "`temp`".

5. In addition to the previous suggestion, libraries could be structured

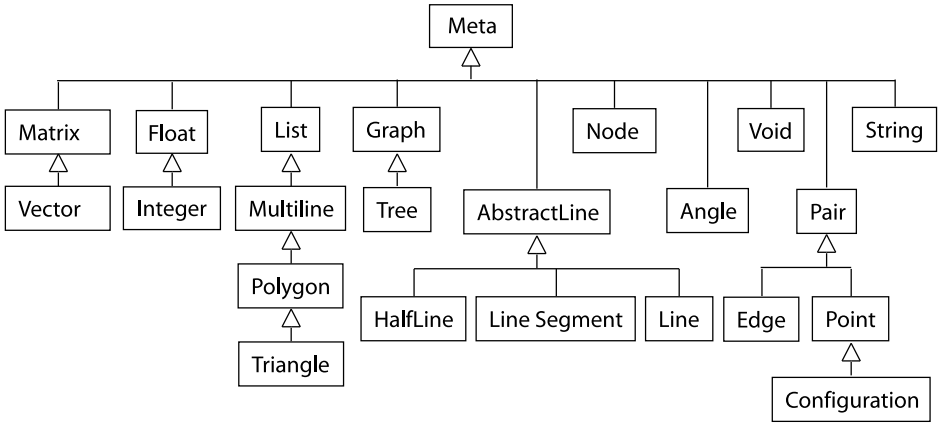


Figure 4.1: A sketch of possible future extensions to the type hierarchy.

in a hierarchy of namespaces. Moreover,the MPLL standard library alone could be organized in domain specific namespaces. For instance a namespace `mpll::polygon` could store functions like `polygon`, `area`, `com`, while a namespace `mpll::point` would contain functions such as `getX`, `getY`, etc.

6. Reference systems still await implementation. While it is sensible to provide some default reference systems it is worth to consider implementing the means to allow their full customization. Currently the MPLL type `Point` can only store `Angles`. For certain applications it might be desirable to allow the definition of points in an Euclidean vector space. It is even imaginable to mix both systems. A point described by a tuple $(x, y) \in$`Angle`$\times\Re$ for example would reflect the location on the surface of a cylinder. While not every such mixture is useful some may be.

7. Points could be extended to hold a certain number of values. Two to three dimensional vectors are quite commonly used data structures in two dimensional geometry. Also, matrices could prove to be a useful addition as discussed in A.2. They are commonly used to describe affine transformations such as rotation, scaling, and movement.

8. In 3.2.4 several additional MPLL types have been suggested. Fig. 4.1 illustrates a possible extension to the type hierarchy.

9. In A.5 it has been pointed out that there is no way of throwing an exception or otherwise handle unresolvable issues during the execution of a custom MPLL function. A very basic but effective solution could be an `error` function. Its sole purpose would be to cause an exception and thus forcefully terminate the execution. This function would be required to declare a return type which is compatible with the functions return type. Otherwise it would not be possible to integrate it into a custom MPLL function declaration.

Appendix A

A.1 Checklist

This chapter is a tutorial. It has been added with the intention of helping to understand the provided software. Furthermore, it may serve as a guide to future extensions. Adding new MPLL types and MPLL functions can be done stepwise. The following listing may serve as a checklist. To better illustrate these steps, excerpts of the source code of the respective parts are provided.

1. Add a C++ implementation of the data structure to store and access the required information if necessary. Custom implementation like these are commonly stored in the C++ namespace `dataType`.

2. Extend the enumeration `TypeId` in file `Type.h` by another entry denoting the *ID* of the new type.

3. Make the new type available as an *MPLL value*. Extend the union `MPLLValue` found in the file `Type.h`. Associate the C++ data structure with an appropriate new name.

4. Add a new type class in `Type.h` and `Type.cpp` by extending and overriding the appropriate type class, e.g. `Type`, `BasicType`, `BTNumber`, etc..

5. Add a line to the `initialize()` function in `Type.cpp` to make it available as a reference. After that it can be quickly retrieved by the method `BasicType::getType(TypeId)`. It is important to pay attention of where to add the line. It needs to antecede its parent type.

6. Add a new class to `MPLL.h` and `MPLL.cpp` respectively, corresponding to the `Type` of function that is to be implemented. The parameters of the constructor should include `Type`s which correspond to the parameters of the intended MPLL function.

   (a) Manually add the expected types to the vector attribute `expectedType`.

   (b) Add the `Type` parameter(s) to the vector attribute `ArgumentTypes`.

   (c) Invoke the function `checkCompatibility` to verify the compatibility of the types. Add the generated error if non-empty.

   (d) Set the attribute `resultType` to the required return type. NOTE: This `Type` may vary depending on the actual `Types` of the input.

7. Override the `apply(...)` function of the class discussed in point 6.

   (a) Retrieve the `vector` of the automatically converted types by calling the function `getCastedTypes`.

   (b) Determine the the result in form of a `MPLLValue` and push it together with its result type onto the stack.

8. If necessary extend the file `Scanner.l` to define keywords in form of new tokens.

9. Add a declaration of the new token(s) in file `Parser.y` and extend or add new syntax rules to incorporate hard coded functions or constructors. The action should usually include the instantiation of a `Token` object. In case of an error the `Token` is to be deleted or pushed onto the stack otherwise.

A.2 Implementing a matrix type

After having established this step-by-step guide it is time to illustrate this by using a more practical example. Based on the current state of MPLL this section sketches out the implementation of a new type `Matrix`. A matrix is a two dimensional array of numbers. In computational geometry they are commonly used to describe affine transformations such as rotation, scaling or

dilation, mirroring and movement. It requires a few operations like matrix-multiplication, multiplication with a scalar, calculation of determinants or inverse matrices, etc. The following sketch demonstrates the necessary steps to implement such features using said check-list.

1. Evidently, a matrix can be represented by a two-dimensional array of floats or doubles. Therefore it does not necessarily require an own class. Nevertheless, there are rather complex and time consuming operations such as matrix multiplication. Given the right MPLL functions this could be realized purely in MPLL, however, this would be too inefficient. That is why their implementations should be realized in C++. For that reason a matrix should better be assigned an own class. A.1 illustrates a rough sketch of that class. The function names should be self explanatory.

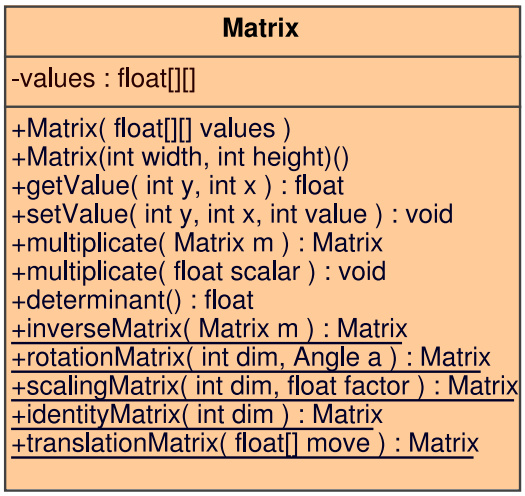


Figure A.1: uml

2. Now the enumeration `typeId` needs to be extended. The result could look like this:

```
enum TypeId {TMatrix, TPoly, TInteger, TAngle, TPoint,
2  TConfiguration, TList, TTime, TFloat, TBool, TSide,
   TPosNeg, TUpDown, TSDVersion, TIntvRegion, TPointRegion,
```

```
4   THull, TFuzzify, TInclusion, TInterval, TString, TVoid,
    TCompound, TPolygon, TMultiline, TMeta};
```

3. In addition `MPLLValue` needs to be extended and might look like this:

```
union MPLLValue {
2     DataType::Matrix*                 lMatrix;
      DataType::Angle*                  lAngle;
4     DataType::Point*                  lPoint;
      DataType::Configuration*          lConfiguration;
6     vector<DataType::Point*>*         lPolygon;
      vector<DataType::Point*>*         lMultiline;
8     vector<pair <Type*, MPLLValue> >* lList;
      Rt*                               Time;
10    FuTI::Interval*                   Interval;
      Function*                         lFunction;
12    int                               Integer;
      float                             Float;
14    bool                              Bool;
      string*                           String;
16  };
```

4. It is sensible to view `Matrix` as a basic type. Therefore the type class for the matrix is named `BTMatrix`. It extends the class `BasicType`. The header file may now contain something like this:

```
class BTMatrix :public BasicType  {
2   public:
      BTMatrix(const string& name, const string& description,
4       TypeId id, bool temp=false) : BasicType(name,
          description,id,NULL, BasicType::getType(TMeta),temp) {}
6     static void initialize();
      static BasicType* parse(const string& token,
8       MPLLValue& value, string& error);
      MPLLValue parse(const string& token, string& error) const;
10    string toString(MPLLValue value) const;
  };
```

The class refers to the super constructor in `BasicType`. The argument specified by `NULL` refers to its attribute `genericType`. It means that it is not part of the generics. It is worth considering this nonetheless, as it is imaginable to define a `Matrix<Integer>`, `Matrix<Float>` or even a `Matrix<Polygon>`. The latter does not contain number types and as such rules out mathematical operations like matrix-multiplications. It would be merely another fancy data type storing two-dimensional lists. In fact it may be even a good idea to define the parent type as a `List<List<Float>>` or a `List<Float>` depending on whether a nested or flat representation of the matrix is desired. Then again the missing functionality could be implemented for special types of matrices, e.g. `determinant [ Matrix<Float>) ↦ Float ]`. For now the parent remains `TMeta` given by the parameter `BasicType::getType(TMeta)`.

5. In the file `Type.cpp` the function `initialize(...)` needs to be extended. It should call the `initialize` function of `BTMatrix`. As its parent type is `BTMeta` it must not be declared before the line `BTMeta::initialize();`. The result could look like this:

```
void BasicType::initialize() {
    BTMeta::initialize();
    BTMatrix::initialize();
    BTVoid::initialize();
    BTNumber::initialize();
    BTAngle::initialize();
    BTPoint::initialize();
    BTConfiguration::initialize();
    BTList::initialize();
    BTMultiline::initialize();
    BTPolygon::initialize();
    BTEnum::initialize();
    BTInterval::initialize();
    asLabel = false;
}
```

6. The class `Matrix` in `MPLL.h` / `MPLL.cpp` needs to be created. Commonly these classes inherit from Application. If it has not been done before, it is necessary now to assess how the MPLL constructor function of the Matrix should look like in order to determine what parameters

are required. Because the Type `List` is available it is reasonable to use it as a parameter. Furthermore the dimensions of the matrix need to be specified. The possible signature might be `Matrix [ Integer * Integer * List<Float> ↦ Matrix ]`. NOTE: If Matrix was to be a generic as discussed before, the signature could be `Matrix [ Integer * Integer * List<T> ↦ Matrix<T> ]`. The header file should now contain this:

```
/* ********************** Matrix ************************ */

class Matrix : public Application {
public:
  Matrix(Type* type1, Type* type2, Type* type3,
     Function* FCT);
  string OperatorName() const {return "newAngle";};
  void apply(int& pc,
     vector<pair<Type*,MPLLValue> >& parameters,
     vector<pair<Type*,MPLLValue> >& stack);
};
```

(a) The expected type for the MPLL constructor function should be stored in the attribute `expectedTypes`. In this case that would require following lines in MPLL.cpp:

```
Type* tInteger = BasicType::getType(TInteger);
Type* tFloat = BasicType::getType(TFloat);
Type* tFloatList = BasicType::copyBasicType(
   BasicType::getType(TList));
tFloatList.setGenericType(tFloat);
expectedTypes.push_back(tInteger);
expectedTypes.push_back(tInteger);
expectedTypes.push_back(tFloatList);
```

First, the instances of `Float` and `Integer` are retrieved by the convenience function `BasicType::getType(TypeId)` as a further reference. The same is done for `List`. In order to avoid side effects the list type gets copied. Its type is expected to be `List<Float>`. Because of that the type textttFloat is set as the generic type

of the copy of the list. Note that this is the reason why it does not suffice to provide the `typeIds`. Then the types are pushed into the stack. It represents the first two parameters to be the integers describing the dimensions or rather the width and height of the matrix. The third parameter is the list of floats storing the values of the intended matrix. Note that if the matrix was to be a generic, a new instance of a `PolyType` could bee used instead of `Float`. This would signal that the expected List was of type `List<S>`. The changes would look like this:

```
PolyType polyType("S");
Type* tPolyList = BasicType::copyBasicType(
   BasicType::getType(TList));
tPolyList.setGenericType(polyType);
...
expectedTypes.push_back(tPolyList);
```

(b) The Types that are provided as the constructor's parameter are added to the attribute `ArgumentTypes` like this:

```
ArgumentTypes.push_back(type1);
ArgumentTypes.push_back(type2);
ArgumentTypes.push_back(type3);
```

(c) It needs to be verified that the argument types and the expected types are compatible. For that purpose the function `bool Application::checkCompatibility(string& name)` is used.

```
string msg = "";
if (! checkCompatibility(msg)) {
   FCT->addError("Matrix " + msg);
}
```

These few lines ensure that the parameters match. Otherwise the function stores an error message, which then is added to the function which will later display it properly.

(d) Finally the result type of the MPLL function needs to be specified. This can be done in a single line:

```
resultType = BasicType::getType(TMatrix);
```

On the other hand, a generic `Matrix` would need to return the template represented by the PolyType as well. The return type would need to be specified in the same fashion as it has been done with the type of the expected list.

```
BasicType* tMatrix = BasicType::getType(TMatrix);
tMatrix = BasicType::copyBasicType(tMatrix);
tMatrix.setGenericType(polyType);
resultType = tMatrix;
```

7. As it shows, the constructor is responsible for verifying only the types. It does not handle any actual values during *parse time*. The whole expression needs to be evaluated first as MPLL function calls may be nested. If this succeeds the `apply` function is invoked during *run time*. This function retrieves the actual parameter values from the stack. Like the Matrix type they were parsed and evaluated before as these are nested in the MPLL function call of the matrix. Depending on these values the value of the matrix can be determined. The resulting code may look like this:

```
void Matrix::apply(int& pc, vector<pair<Type*, MPLLValue> >&
     parameters, vector <pair<Type*, MPLLValue> >& stack) {
  vector<MPLLValue> castedTypes = getCastedTypes(pc, stack);
  int width  = castedTypes.at(0).Integer;
  int height = castedTypes.at(1).Integer;
  List list  = castedTypes.at(2).lList;
  if (width * height != list.size()) {
     throw ParserException((pc-1), "Matrix::apply",
        "Capacity does not match number of provided
           elements.");
  }
  float[width][height] array;
  for(unsigned int i=0; i<list.size(); i++) {
     array[i%width][height/i] = list.at(i).second;
  }
  Matrix* matrix = new Matrix(array);
  MPLLValue result;
```

```
18    result.lMatrix = matrix;
      push2Stack(result,resultType,stack);
20  }
```

First the casted values are retrieved. In case the types are compatible but not identical their corresponding values get casted to until they match the expected type. As opposed to the legacy code there is no need for manually checking for valid types or manually converting the required values which is a great benefit of the introduced type hierarchy. That means that a list of integers is accepted as well but then automatically casted to a list of floats. This example shows the acceptable use of a `ParserException`. Even though many errors can be detected during *parse time*, it may happen that the actual content of the values itself may cause problems. In this example that is the case when the number of floats in the list does not match the capacity defined by the width and height values. Finally, the entries in the list get read and stored into an array which is used to create an instance of matrix. It is then stored as an `MPLLValue` and pushed onto the stack.

8. In order to use the newly implemented type it is necessary to provide the syntax definition for calling the constructor. The first step in doing is to identify the keyword `"matrix"` with a token. Here it is called `MPLL_MATRIX`. In file `Scanner.l` the following line needs to be inserted.

```
"matrix"          {*MPLLColumn += 6; return MPLL_MATRIX;}
```

9. The aim is to describe the syntax of the MPLL constructor function for the type `Matrix`. Therefore it is necessary to extend the grammar rules of `expression` in the file *Parser.y*. Following the keyword "*matrix*" represented by the token `MPLL_MATRIX` three parameters are to be specified. The types of these three are then forwarded to the constructor of Matrix in the namespace MPLL as explained in point 6. If this succeeds without any error, the Application with the variable name `Op` is then pushed into the stack for further processing.

```
| MPLL_MATRIX '(' expression',' expression',' expression ')' {
2   MPLL::Matrix* Op = new MPLL::Matrix($<type>1, $<type>2,
      $<type>3, MPLL_function);
```

```
4   if(MPLL_function->noError()) {
      MPLL_function->push_back(Op);
6     $<type>$ = Op->resultType;
    } else delete Op;
8 }
```

Additionally said token needs to be declared as a value in the same file.

```
%token <value>   MPLL_CAST
```

Now the foundation for creating the MPLL type Matrix has been established. Now the new type can be tested, for example, by using the `query` function in `MPLLVM`. The expression `Matrix(2,3,List(1,2,3,4,5,6))` would be resolved to

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

Instead of calling `query` manually a bunch of test cases can be added to the `MPLL test suite` in file `stdlib.query`. Possible entries could look like this:

```
 'matrix(2,3,List(1,2,3,4,5,6))' '+'
2 'matrix(6,1,List(1,2,3,4,5,6))' '+'
 'matrix(2,2,List(1.5,2.3,3,4.9))' '+'
4 'matrix(2,1,List(Angle(),Angle()))' '-'
 'matrix(2,3,List(1,2,3,4,5))' '-'
6 'matrix(2,3,List(1,2,3,4,5,6,7))' '-'
```

While the first three test cases should execute fine, the other three are expected to fail. The fourth case provides a list of angles which causes compatibility issues. The list in the 5th and 6th case does provide too few and too many elements respectively.

There is a lack of MPLL functions that access the type `Matrix`. In the following three examples of future MPLL functions are given. The first one is going to be the mpll functions `multiplicate` that realizes matrix multiplication. It takes two matrices as an argument and returns a new matrix. As this is a rather complex algorithm it is assumed that the data structure `Matrix` already implemented this as shown in the uml diagram in Fig. A.1.

Therefore this function is be hard coded. The second and third example demonstrates the use of a custom MPLL function. One is called `get` and requires two integers representing the position of a given entry in the matrix. Thus it is supposed to return a Float. Furthermore, the type hierarchy is expanded so `Matrix` is a direct subtype of `List` which in turn enables the use of preexisting functions. The last example illustrates another custom MPLL function that realizes the addition of two matrices in the next secion.

A.3 Implementing matrix multiplication

In this section the same checklist is abode. Steps one to five can be skipped. There is nothing to do as no new data structure is required. The only thing left are the modifications that are need to implement the function.

6. All hard coded functions that can notionally be assigned to the type `Matrix` are unified in one class `MatrixPredicate` to avoid cluttering code with classes. This is obviously not necessary but a good way of maintaining the code's readability. Only two parameters of type `Type*` are required. Both should be resolved to a `Matrix` later on. They both represent matrices that need to be multiplicated. Furthermore it is a good idea to introduce another parameter of type `int`. It denotes the *ID* or *Op-code* of the function. If further functions are to be incorporated, a *switch-case statement* suffices to determine the actual function. For that reason this example includes said statement even though it is not necessary yet. Furthermore additional functions may vary in their number of arguments. For that reason it may become necessary to provide another integer parameter which communicates the number of parameters to be stored in the vector `argumentTypes`.

   (a) The expected types are determined upon reading the *Op-code* which implicates the intended function. For this function these are two arguments of type Matrix. If the type matrix was a *generic* it would be necessary to specify both of them to be of type `Matrix<Float>` as arithmetic operations are necessary.

   (b) The argument types are pushed back as usual.

   (c) The same applies for the compatibility check. In fact this part can always be reused without any further alteration.

(d) Depending on the Op-code the result type needs to be determined. In this case this is again a matrix.

7. Now the function `apply` needs to be overridden.

   (a) The casted types need to be generated. Like in 6b the usual code can be copied and pasted.

   (b) Again the Op-code can be consulted to assess what needs to be done. Again in this case there is only one possibility. The two matrices are retrieved from the vector of casted types and forwarded as parameters to the actual data structure to handle the actual calculation. The result gets wrapped in an `MPLLValue` and pushed onto the stack. It is important that both matrices must meet a certain requirement. For a matrix multiplication the width of the first matrix must be equal to the height of the second one. In order to provide another example of the usage of a `ParserException`, this check is included in here. Alternatively it could have been handled by the data structure itself.

8. As there is no other function by the name `multiply` the keyword needs to be identified with a token `MPLL_MATRIX_MULTIPLY`. The implied naming convention is a sensible approach to avoiding obfuscation. It becomes clear that this token belongs to the domain of functions related to the type matrix.

9. Said token is again declared as a value. The definition of `expression` is enriched by another definition. It should require two expressions as arguments in addition to the function name represented by the token. The associated action should use the constructor of `MPLL::MatrixPredicate` to create its instance.

The *MPLL test suite* may be enlarged to test the newly implemented function.

```
'multiplicate(Matrix(1,1,List(-3.7)),Matrix(1,2,List(10,1)))' '+'
'multiplicate(Matrix(2,3,List(1,2,3,4,5,6)),
   Matrix(3,1,List(7,8,9)))' '+'
'multiplicate(Matrix(2,2,List(1,2,3,4)),
   Matrix(3,2,List(5,6,7,8,9,10)))' '-'
```

A.4 Accessing elements in a matrix

The next function to be introduced is the function called `get`. It simply retrieves the element at a specific position. Elements of a matrix are usually given by a pair. Unlike in a coordinate system the position is given by the index of the row first followed by the index of the column. In order to exemplify the type hierarchy, it is demonstrated how to alter the type matrix to become a subtype of list. In addition it is assumed that two functions `width` and `height` are in place. As the name suggests they return the respective size of a given matrix. Thus, it is possible to utilize the list function `element`, along with these functions to retrieve the requested entry. As a motivation an example MPLL query is given first.

```
get(1,2,Matrix(3,3,List(1,2,3,4,5,6,7,8,9)))
```

This should return the element at row 2 and column 3 as the index starts at zero. Obviously that is the float value `6` in this case. Anticipating that `Matrix` is a subtype of list, which is established shortly, `get` can be implemented as follows:

```
get(Integer row, Integer col, Matrix m) =
2   element(m, width(m)*row + col)
```

The type of function `element` is `List<T> * Integer` $\mapsto$ `T`. Because argument `m` is of type `Matrix` which is a subtype of `List`, `m` should automatically be casted to `List<Float>` and thus the return value should be of type Float.

The actual adaption of Matrix is rather simple. It had been mentioned before that there is an argument of the superclass `BasicType` that specifies the parent type. As a consequence the class `BTMatrix` needs to be slightly modified.

```
class BTMatrix :public BasicType  {
2  public:
     BTMatrix(const string& name, const string& description,
4       TypeId id, bool temp=false) : BasicType(name,description,
          id,NULL, new BTList(
6            static_cast<BTList*>(BasicType::getType(TList)),
```

```
                BasicType::getType(TFloat)
8           )
        ,temp) {}
10    ...
```

Admittedly this may look chaotic but it is necessary. The parent type needs to be specified in the head of the function. As such it must be instantiated there as well by using a convenience constructor of `BTList`. The second argument specifies that the list stores a `Float`. This is already everything that needs to be changed concerning the type hierarchy. During *parse time* the example MPLL statement already executes without any further troubles.

Concerning the handling of actual MPLL values the previously mentioned function `getCastedTypes` provides a vector of type converted values. This function again delegates this task to another function called `MPLLValue Application::castType(Type* expected, pair<Type*, MPLLValue> actual)`. It contains a switch-case statement to determine the current type of the value and then, provides a manual translation to the parent type. A `Polygon` for instance gets converted to a `Multiline`, a `Multiline` to a `List` of `Points` and so forth. For `Matrix` to properly act as a subtype of `List`, a case needs to be added where said conversion takes place.

```
...
2 case TMatrix:
    vector<pair<Type*, MPLLValue> >* pointList =
4       new vector<pair<Type*, MPLLValue> >();
    Type* pointType = BasicType::getType(TPoint);
6   Matrix* m = actual.second.lMatrix
    for (unsigned int i = 0; i<m.getWidth()*m.getHeight(); i++) {
8       MPLLValue v;
        int x = i % m.getWidth();
10      int y = i / getWidth();
        v.lPoint = m.getValue(y,x);
12      pointList->push_back(make_pair<Type*,
           MPLLValue>(pointType,v));
14  }
    newActualValue.lList = pointList;
16  break;
...
```

As the code shows the elements of the matrix are read out and pushed onto a vector. This vector is then wrapped into an MPLLValue as list which contains points.

A.5 Implementing matrix addition

The final example shows how to implement matrix addition. Two matrices of same width and height can be added by pair wise addition of their corresponding elements. Even though it may not be the most efficient method this example relies on a custom MPLL function.

```
Matrix:add_helper(List<Point> m1, List<Point> m2, Integer width,
2   Integer height, List<Point> list) =
    if (empty(m1)) then
4      return Matrix(width, height, list)
    else add_helper(tail(m1), tail(m2), width, height,
6      append(head(m1)+head(m2),list))

8 add(Matrix m1, Matrix m2) =
    add_helper(m1, m2, width(m1), height(m1), List<Point>())
```

MPLL function `add` uses a helper function `add_helper` providing the width and height of desired result matrix. An additional argument is a list of points that serves as an *accumulator*. The helper function adds the first elements of both matrices by the use of `head`. It is a list function which is now available to Matrix as it is a subtype of `List`. The sum of the elements is then concatenated to the accumulator list. The function calls itself recursively, passing on the tails of both lists every time. As soon as the lists are empty the result matrix is generated based on the width, height and the point list. It may be noted that this function does not verify the width and height of both matrices. In case the first matrix contains more elements than the second one, an exception is thrown due to an *index-out-of-bounds exception*. On the other hand, the first matrix may contain less elements which would not throw any exceptions. For a suggestion to solve this issue see 7.

Appendix B

```
**********
* Basics *
**********

'PI = 3.14159265'

*********
* Maths *
*********

'root(Float b, Float e) = pow(b, 1.0/e)'
'sqr(Float f) = pow(f, 2.0)'
'sqrt(Float f) = root(f, 2.0)'

*********
* Angle *
*********

'Angle(Float v, Float a, Float b) = newAngle(v,a,b)'
'Real = 0'
'Grd = 1'
'Deg = 2'
'Rad = 3'
'isNegative(Angle a) = (grad(a) < 0)'
'degToGrd(Float i) = i*10/9'
'degToRad(Float i) = i*PI()/180'
'grdToDeg(Float i) = i*9/10'
'grdToRad(Float i) = i*PI()/200'
'radToGrd(Float i) = i*200/PI()'
'radToDeg(Float i) = i*180/PI()'
'degree(Angle a) = grdToDeg(grad(a))'
'radian(Angle a) = grdToRad(grad(a))'
```

```
  'modulus(Angle a) = max(a)-min(a)'
34 'defMod(Integer type) =
  case   (type == Real()):0,
36       (type == Grd()):400,
         (type == Deg()):360,
38       (type == Rad()):2*PI()
  else 0'
40 'Angle(Float f, Integer type) = newAngle(f,-defMod(type),defMod(type))'
  'Angle() = Angle(0.0,Grd())'
42 'AngleReal(Float f) = Angle(f,Real())'
  'AngleReal(Float angle, Float mod) = Angle(angle, -mod, -mod)'
44 'AngleGrd(Float f) = Angle(f,Grd())'
  'AngleGrd(Float angle, Float mod) = AngleReal(angle, mod)'
46 'AngleDeg(Float f) = Angle(f,Deg())'
  'AngleDeg(Float angle, Float mod) = AngleGrd(degToGrd(angle),
48   degToGrd(mod))'
  'AngleRad(Float f) = Angle(f,Rad())'
50 'AngleRad(Float angle, Float mod) = AngleGrd(radToGrd(angle),
     radToGrd(mod))'
52 'equals(Angle a, Angle b) = (grad(a)==grad(b)) and (min(a)==min(b)) and
     (max(a)==max(b))'
54
  *********
56 * Point *
  *********
58
  'Point(Angle x, Angle y) = newPoint(x,y)'
60 'Point() = Point(Angle(),Angle())'
  'Point(Float x, Float y) = Point(AngleGrd(x),AngleGrd(y))'
62 'distance(Point p1, Point p2) =
         let deltaX = grad(getX(p2)) - grad(getX(p1)) in
64       let deltaY = grad(getY(p2)) - grad(getY(p1)) in
         sqrt(deltaX + deltaY)'
66 'vpl(Point p1, Point p2, Point p3) = (p2-p1)*(p3-p1)'
  'isLeft    (Point p1, Point p2, Point p3) = (vpl(p1,p2,p3) >  0)'
68 'isRight   (Point p1, Point p2, Point p3) = (vpl(p1,p2,p3) <  0)'
  'isColinear(Point p1, Point p2, Point p3) = (vpl(p1,p2,p3) == 0)'
70 'equals(Point a, Point b) = equals(getX(a),getX(b)) and
     equals(getY(a),getY(b))'
72
  ********
74 * List *
  ********
76
  'head(List<T> x) = element(x,0)'
```

```
'empty(List l) = (size(l) == 0)'
'append2(T h, List<T> t) = append(newList(h),t)'
'List<T>:prefix(List<T> list, Integer index) =
        if (index == 0)
        then newList(head(list))
        else append2(head(list), prefix(tail(list),index-1))'
'List<T>:suffix(List<T> list, Integer index) =
        if (index == 0)
        then list
        else suffix(tail(list), index-1)'
'List<T>:reverse(List list) = append(reverse(tail(list)),
   newList(head(list)))'
'subList(List list, Integer from, Integer to) =
   suffix(prefix(list,to),from)'
'remove(List list, Integer index) =
        if (index==0)
        then tail(list)
        else
           if (index==(size(index)-1))
           then prefix(list, (size(list)-2))
           else append( prefix(list,(index-1)) , suffix(list,(index+1)) )'

**********************
* Configuraton *
**********************

'Configuration(Angle a, Angle b, Angle c, Bool d) =
   newConfiguration(a,b,c,d)'
'Configuration(Angle b, Angle x, Angle y) = Configuration(b,x,y,true)'
'Configuration(Angle x, Angle y) = Configuration(Angle(),x,y,false)'
'Configuration(Angle b, Point p, Bool t) =
   Configuration(b,getX(p),getY(p),t)'
'Configuration(Angle b, Point p) = Configuration(b,p,true)'
'Configuration(Angle b) = Configuration(b,Point(),true)'
'Configuration() = Configuration(Angle(),Point())'
'Configuration(Point p) = Configuration(getX(p),getY(p))'
'Configuration(Float x, Float y) = Configuration(Point(x,y))'


*************
* Multiline *
*************

'Multiline(List<Point> list) = newMultiline(list)'
```

```
  'isPolygon(Multiline ml) = (size(ml) > 3) and
124    equals(head(ml),element(ml,size(ml)-1))'

126 ***********
  * Polygon *
128 ***********

130 'Polygon(List<Point> list) = newPolygon(list)'
  'isTriangle(Polygon p) = (size(p)==3)'
132 'Float:area2(List<Polygon> lp)'

134 'area(Polygon p) =
          if (isTriangle(p))
136       then vpl(head(p), element(p,1), element(p,2))/2.0
          else let tri=triangulate(p) in area2(tri)'
138
  'area2(List<Polygon> lp) =
140       if empty(lp)
          then 0.0
142       else area(head(lp)) + area2(tail(lp))'

144 'Point:com2(List<Polygon> lp)'
  'Point:com3(List<Polygon> lp)'
146
  'com(Polygon p) =
148       if (isTriangle(p))
          then (head(p) + element(p,1) + element(p,2))*(1.0/3.0)
150       else let tri=triangulate(p) in com2(tri)'

152 'com2(List<Polygon> lp) = com3(lp)*(1.0/size(lp))'

154 'com3(List<Polygon> lp) =
          if empty(lp)
156       then Point()
          else com(head(lp)) + com3(tail(lp))'
158
  List<E>:map(T->E f, List<T> list) =
160       if (empty(list))
          then newList()
162       else append(f(head(list),map(f,tail(list)))
```

Index

Bibliography

[1] Hans Jürgen Ohlbach. Gets - a specification language for geo-temporal notions. In *Proceedings of 29th Annual German Conference on Artificial Intelligence, Bremen, Germany (14th–19th June 2006)*, 2006.

[2] Bernhard Lorenz. Multi-paradigm spatial information processing. 2006.

[3] B. Chazelle. Triangulating a simple polygon in linear time. *Disc. and Comp. Geometry*, 6:485–524, 1991.

[4] Franklin & marshall - polygon triangulation: triangulate a monotone polygon. `http://www.fandm.edu/x7743.xml`.

[5] Edgar-Philipp Stoffel. A research framework for graph theory in routing applications. Diplomarbeit/diploma thesis, Institute of Computer Science, LMU, Munich, 2005.