Universität Bremen
Fachbereich 3 - Mathematik und Informatik

A comparing case study on the formal verification process between implementations in imperative and functional programming languages

Masterarbeit

Vorgelegt von:	Tobias Brandt
E-Mail:	to_br@uni-bremen.de
Matrikelnummer:	2914844
Studiengang:	Informatik, M.Sc.
Erstgutachter:	Prof. Dr. Christoph Lüth
Zweitgutachter:	Prof. Dr. Udo Frese

Bremen, September 2018

Abstract. In safety-critical areas the need for formal verification of software systems is rising despite being a costly task. Likewise, the popularity of concepts of functional programming languages has increased. Since the functional programming paradigm is very closely related to mathematics, the question is raised if functional programming can be beneficial to the use of formal methods. This thesis will investigate this question by performing a case study. In the case study an existing algorithm with an imperative implementation and a formal verification is reimplemented in a pure functional language. After that, the new implementation is formally verified. From there, a comparison between both implementations and their verification processes is drawn to explore if a functional programming language may accelerate the verification process while still being applicable in production environments.

Eidesstattliche Erklaerung

Ich versichere, die Masterarbeit selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Bremen, 17.09.2018

_____

(Tobias Brandt)

Table of Contents

1 Introduction

1.1 Motivation

In recent years the idea of autonomous vehicles has become more popular, e.g. there is the company *Waymo*, whose goal it is to develop *self driving cars* for the general public [26]. Since malfunctions in autonomous vehicles might pose a severe danger to human lives, correctness of software systems becomes evermore important. Thus, the need for applicable formal verification methods is rising. However, formal verification of software systems is a costly task, since the specification of such systems is often formalized in the language of mathematics and requires a complex series of non-trivial mathematical proofs.
Meanwhile, functional programming has grown in popularity as well. Although there are very few predominantly functional programming languages in the top 50 of the *TIOBE Index* [21], there are features originating from declarative languages that find more adoption in imperative languages. An example is the release of *Java 8*, which introduced *lambda expressions* and the *Stream API*, which enables support for higher-order functions [16]. Functions in pure functional programming languages are referential transparent, which means that all applications of a function can be replaced with the corresponding value of the function application without changing the semantics of the program [14, p. 78-79]. By definition, this property is shared with mathematical functions as well. This is raising the question if the cost of formal methods, i.e. time of the verification process, can be decreased by implementing the desired program in a pure functional programming language in contrast to an imperative one. The present thesis will explore this question.

1.2 Scope

This thesis will investigate if an implementation in a functional programming language can accelerate the time of the verification process, because of the close relations between functional programming and mathematics. To explore if a functional language can be beneficial in this regard, a case study will be performed. The case study will operate on an existing algorithm, which has a formal specification, is implemented in an imperative programming language and is formally verified already. The existing specification will be kept and will be used to develop a new implementation in a pure functional programming language. The implementation will then be formally verified. Following that, a comparison will be drawn between the existing imperative implementation and the new functional one. The comparison will include the time of the verification processes as well as the runtime performance of both implementations.

1.3 Outline

The thesis at hand is structured as follows: chapter 2 offers a brief introduction into the selected case study. Section 2.1 provides an informal overview of the case study, while section 2.2 is adding

the formal mathematical description. The last section of this chapter, section 2.3, describes the actual implementation of the algorithm provided by the case study as well as a description of the formal verification process that was involved.
Chapter 3 describes the functional implementation that was developed (section 3.1) and its formal verification process (section 3.2). Additionally, section 3.3 presents a performance comparison between the original implementation and the functional one.
Chapter 4 discusses the findings, provides an outlook into possible future research topics and concludes this thesis.

2 Case Study Description

As stated previously, the aim of this thesis is to show that programs written in a pure functional language are significantly easier to verify. Therefore, this thesis will provide a case study, which draws a comparison between the verification process of an imperative and a functional implementation of the same algorithm. There were three main requirements that the domain of the case study needed to meet. First, the program had to be formally specified to compare the verification result against a desired outcome. Second, in order to state that a functional implementation is indeed faster to verify than an imperative implementation, a verified imperative implementation was needed. Lastly, a sufficient insight into the verification process was necessary, such that metrics to affirm the hypothesis could be derived. All these requirements were satisfied by an algorithm presented in the paper *Guaranteeing functional safety: design for provability and computer-aided verification* [20].

The paper and its algorithm arose as an outcome of the SAMS project (*Safety Component for Autonomous Mobile Service Robots*), which was a joint project by the *University Bremen*, *Leuze electronic*, lead by the *DFKI Bremen* [3]. The paper describes an algorithm for autonomous driving vehicles, that computes a safety zone around the vehicle based on its current motion. It is therefore a suitable measure against collisions with static objects. To ensure the correctness of the algorithm, the authors specified the algorithm formally and then implemented a version of the algorithm in *C*. Afterwards, the implementation was verified with the help of the interactive theorem prover *Isabelle* [22].
The following chapter will explain said algorithm informally and formally. Finally, the imperative reference implementation in *C* will be described.

2.1 Informal Description

This chapter will provide a brief overview of the safety zone algorithm. A detailed mathematical description will follow in the next chapter.

The input of the algorithm is the current translational and rotational velocity of a vehicle at time t_0. The algorithm is then partitioned into two steps. First, the braking distance is computed. The braking area is expressed as a length alongside a circle s and an arc α. This is a consequence of the simplified braking model, which assumes that the vehicle is always moving either in a straight line or in alongside a circular arc. The braking model is parameterized by a list of previously taken braking measurements for the respective vehicle. These measurements are used to determine an upper bound for the actual braking distance by performing a pairwise linear interpolation between the measurements [25, p. 149].
Second, from s and α the actual safety zone, i.e. the area that covers all points that were touched by the vehicle during braking starting at t_0, is calculated. Figure 1 visualizes this. In theory this would

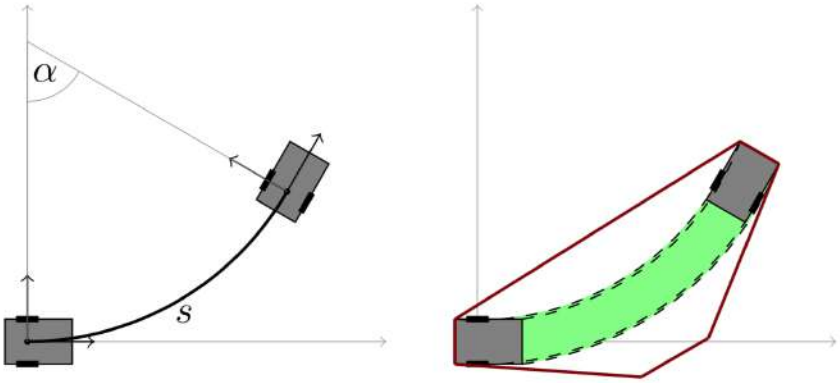


Figure 1: Left: The calculated braking distance expressed as s and α. Right: The computed safety zone, which covers the braking area of the vehicle (green area). Taken from [20].

always calculate the correct safety zone of the vehicle. Unfortunately, in reality, measuring errors are quite common. This would result in incorrect safety zones, which could ultimately lead to the endangerment of human lives. Therefore, the actual algorithm does not take a single translational and rotation velocity, but rather a lower and an upper bound for both velocities. Those represent the intervals around the actual measured velocities. Likewise, the first step of the algorithm does not result in a single (s, α), but also in bounds representing an interval $((s_{min}, s_{max}), (\alpha_{min}, \alpha_{max}))$. The resulting safety zone is the convex hull of all the previous calculated braking areas, which is also the first part of the final result of the algorithm. The second part is an additional buffer radius extending the convex hull, which accounts for numerical errors due to the limitations of floating point arithmetic.

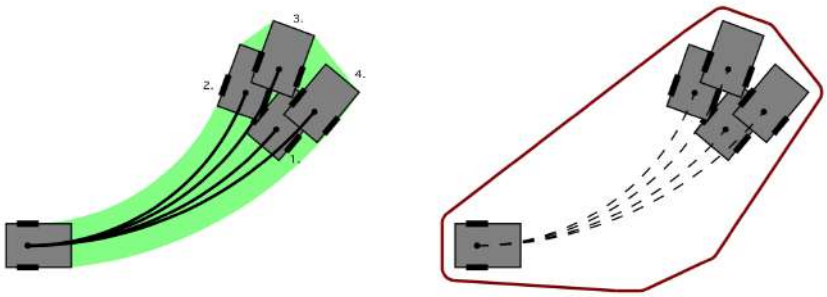


Figure 2: Left: *1.* (s_{min}, α_{min}). *2.* (s_{min}, α_{max}). *3.* (s_{max}, α_{max}). *4.* (s_{max}, α_{min}). The green area represents the union of the actual braking areas of intervals according to the braking model. Right: The computed safety zone of intervals with the additional buffer radius q. Taken from [20].

The right side of figure 2 visualizes the final result of the algorithm including the buffer radius. In the original paper the safety zone is further processed into a laser scan representation [20, p. 19-20]. This step will be omitted in this thesis, so that only the main part of the algorithm is implemented and verified.

2.2 Formal Specification

The following chapter will describe the formal specification of the safety zone algorithm case study. It will only focus on the mathematics that are necessary to implement the algorithm. The proofs and their derivation can be found in the original paper [20, p. 13-20].

The algorithm takes the bounds for the translational velocity $(v_{min}, v_{max}) \in \mathbb{R} \times \mathbb{R}$ as well as the bounds for the rotational velocity $(\omega_{min}, \omega_{max}) \in \mathbb{R} \times \mathbb{R}$ as input.
Further, the first step of the algorithm is configured over variables that are specific to the vehicle. These are the latency Δt, which is the assumed time that the vehicle continues to drive with its current velocity before it starts braking, a sequence of vehicle contour points $\mathfrak{R} = R_1, \ldots, R_n$, and a sequence of straight braking measurements of the vehicle $M = (v_0, s_0), \ldots, (v_m, s_m)$ where v_i refers to a previous taken velocity measurement while driving straight. s_i refers to the corresponding covered distance from the start of the measurement to standstill. It is assumed that the braking measurements are in ascending order. It is also assumed that $(v_0, s_0) = (0, 0)$. Additionally, one measurement must always be taken at maximum speed of the vehicle. Because of the ascending order, this is always (v_m, s_m).

The result of the first step is the bounds of the distance alongside a circle $(s_{min}, s_{max}) \in \mathbb{R} \times \mathbb{R}$ and the bounds for the arc $(\alpha_{min}, \alpha_{max}) \in \mathbb{R} \times \mathbb{R}$ (see Figure 2) with:

$$s_{min} = \min_c s^c \tag{1}$$

$$s_{max} = \max_c s^c \tag{2}$$

$$\alpha_{min} = \min_c \alpha^c \tag{3}$$

$$\alpha_{max} = \max_c \alpha^c \tag{4}$$

All possible (s^c, α^c) are the result of the braking model function $BM\Delta t$, such that $(s^c, \alpha^c) = BM\Delta t(v^c, \omega^c)$.

$$(s^c, \alpha^c) \in \left\{ BM\Delta t(v^c, w^c) \mid (v^c, w^c) \in V \times \Omega \right\} \tag{5}$$

The input of the braking model function is the calculated candidates $(v^c, \omega^c) \in V \times \Omega$, which depend on the input of the algorithm (v_{min}, v_{max}) and $(\omega_{min}, \omega_{max})$

with

$$V = \begin{cases} \{v_{min}, 0, v_{max}\} & 0 \in [v_{min}, v_{max}] \\ \{v_{min}, v_{max}\} & 0 \notin [v_{min}, v_{max}] \end{cases} \tag{6}$$

$$\Omega = \begin{cases} \{\omega_{min}, 0, \omega_{max}\} & 0 \in [\omega_{min}, \omega_{max}] \\ \{\omega_{min}, \omega_{max}\} & 0 \notin [\omega_{min}, \omega_{max}] \end{cases} \tag{7}$$

The algorithm only considers the candidates as input to the braking model function, because all other $(v', \omega') \in [v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}] - V \times \Omega$ cannot result in an extreme in the braking model function and can therefore be neglected.
The braking model function is defined in (8) where $\hat{v}_s$ is the upper bound of the equivalent straight velocity for a curved motion (v, ω) and $\hat{s}(\hat{v}_s)$ is the corresponding upper bound for the braking distance.

$$BM\Delta t(v, \omega) = \left(\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s} + \Delta t \right) \begin{pmatrix} v \\ \omega \end{pmatrix} \tag{8}$$

$$\hat{v}_s = \sqrt{v^2 + D^2\omega^2} \tag{9}$$

with

$$D = \max_i |R_i| \tag{10}$$

$$\hat{s}(v) = \begin{cases} \frac{s_1}{v_1} v & \text{if } v \leq v_1 \\ \frac{s_m}{v_m^3} v^3 & \text{if } v \geq v_m \\ s_{j-1} + \frac{s_j - s_{j-1}}{v_j - v_{j-1}} (v - v_{j-1}) & \text{otherwise} \end{cases} \tag{11}$$

where j is the index such that $v_{j-1} \leq v \leq v_j$. This concludes the specification of the first step of the algorithm.

The input of the second step of the algorithm is the distance and the arc from the previous calculated bounds $(s, \alpha) \in I$ with $I = \{s_{min}, s_{max}\} \times \{\alpha_{min}, \alpha_{max}\}$. The result is a sequence of points, such that their convex hull is an upper bound approximation of the actual braking area for all (s, α) and an additional buffer radius q, which is added on top of the safety zone.

The final result of the algorithm is captured in the function H, where the first component is

the sequence of convex hull points and the second component is the buffer radius.

$$H(s_{min}, s_{max}, \alpha_{min}, \alpha_{max}) = ([[[P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]^{L-1}_{j=0}]^n_{i=0}]^{s_{max}}_{s_{min}}]^{\alpha_{max}}_{\alpha_{min}}, q) \tag{12}$$

The sequence of convex hull points is not the exact braking area, but rather a conservative approximation. The approximation is done by combining the approximations of each trajectory of each contour point of the vehicle. There are multiple ways of approximating the area of a single trajectory. Figure 3 shows four possible alternatives, which are all supersets of that respective trajectory. In the original paper a generalized version of figure 3(d) is used by making the number of subarcs variable.

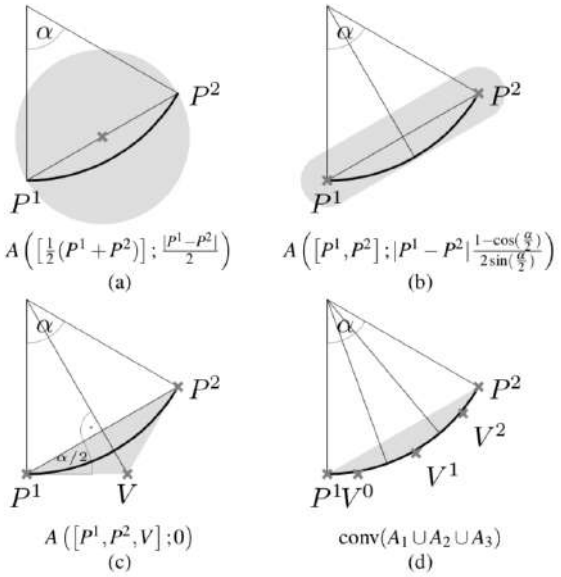


Figure 3: Possible approximation methods of a circular trajectory using *(a)* one, *(b)* two or *(c)* three points to generate the area. In *(d)* the trajectory is first divided into three equal parts; then *(c)* is calculated for each of them. Taken from [20].

$P^1_{i,s,\alpha}$ and $P^2_{i,s,\alpha}$ are the start and end point of a contour point R_i for the braking model (s, α) respectively.

$$P^1_{i,s,\alpha} = R_i \tag{13}$$

$$P^2_{i,s,\alpha} = T(s, \alpha)R_i \tag{14}$$

The transformation matrix $T(s, \alpha)$ is defined as follows. It is assumed that all coordinates are extended by 1 as a component, such that the matrix multiplication is well defined. For readability reasons the additional component will be omitted in this thesis.

$$T(s,\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & s \cdot \operatorname{sinc}\frac{\alpha}{2} \cdot \cos\frac{\alpha}{2} \\ \sin\alpha & \cos\alpha & s \cdot \operatorname{sinc}\frac{\alpha}{2} \cdot \sin\frac{\alpha}{2} \end{pmatrix} \tag{15}$$

with

$$\operatorname{sinc}\phi = \begin{cases} \frac{\sin\phi}{\phi} & \phi \neq 0 \\ 1 & \phi = 0 \end{cases} \tag{16}$$

The variable $V^j_{i,s,\alpha}$ is the generalized version of $V^1 - V^3$ in figure 3(d) and is used to divide the arc into equal parts. The number of approximation points is configured by the variable $L \in \mathbb{N}^+$.

$$V^j_{i,s,\alpha} = T\left(\frac{j \cdot s}{L}, \frac{j \cdot \alpha}{L}\right) \cdot U^3_{i,s,\alpha} \tag{17}$$

$$U^1_{i,s,\alpha} = R_i \tag{18}$$

$$U^2_{i,s,\alpha} = T\left(\frac{s}{L}, \frac{\alpha}{L}\right) \cdot R_i \tag{19}$$

$$U^3_{i,s,\alpha} = U^1_{i,s,\alpha} + Q(\frac{\alpha}{L})\frac{1}{2}(U^2_{i,s,\alpha} - U^1_{i,s,\alpha}) \tag{20}$$

$$Q(\alpha) = \begin{pmatrix} 1 & \tan\frac{\alpha}{2} \\ -\tan\frac{\alpha}{2} & 1 \end{pmatrix} \tag{21}$$

At last, the buffer radius q is defined in (22), which concludes the specification of the algorithm.

$$q = q^A + q^B \tag{22}$$

$$q^A = \frac{1}{6}\left(\frac{\alpha_{max} - \alpha_{min}}{2}\right)^2 \max\{|s_{min}|; |s_{max}|\} \tag{23}$$

$$q^B = \left(1 - \cos\frac{\alpha_{max} - \alpha_{min}}{2}\right) \max_i |R_i| \tag{24}$$

2.3 Implementation and Verification

The case study was implemented in *MISRA-C*, a subset of *C* [13]. The implementation was then verified with a self-developed verification framework [25]. In the verification framework, functions and loops can be annotated with their specifications directly in the *C* code. The annotations are written within *C* comments and therefore do not interfere with the compiler [25, p. 38-40]. The specifications are then checked by the interactive theorem prover *Isabelle* [22].
The final implementation consisted of 5339 lines of code. 2535 lines of those were comments and specifications for the verification framework. The remaining 2804 lines of code were spread over 39 *C* functions, of which 29 where formally verified [20, p. 23]. To verify the algorithm, it was additionally necessary to model the problem domain in *Isabelle*, which resulted in 11 *Isabelle* theory files consisting of about 110 definitions and 510 theorems. This process took a mathematician about 5 months [25, p. 153]. The verification process of the *C* implementation took a *SAMS* project member another 6 months [25, p. 170].
Furthermore, the *SAMS* software was audited by the *TÜV SÜD* and found to be compliant with *IEC EN 61508:3* (*SIL 3*) [4].

3 Verification of the Functional Reimplementation

3.1 Functional Implementation

This chapter will provide a short description of the functional implementation of the case study presented in the previous section. Because of the close resemblance between its syntax and mathematical definitions, *Haskell* [11] was chosen as the implementation language. The complete *Haskell* code, which is directly involved in the algorithm, will be provided. More technical code (e.g. *printing results*) will be omitted, but can be found in Appendix A. The main goal of the implementation was to maintain the close relationship to the mathematical specification. Thus, the implementation aims for a one to one correspondence between a mathematical definition and a *Haskell* function or constant. Instances where this is not the case will be explained in the following.

no equivalent

```
{-# LANGUAGE OverloadedLists #-}
module SafetyZone where

import Data.Vector (Vector, (!))

type Velocity = Float
type Distance = Float
type Time = Float
type SafetyZone = ([Point], BufferRadius)
type BufferRadius = Float
type Point = (Float, Float)

safetyZone :: (Float, Float)→(Float, Float)→SafetyZone
safetyZone vMinMax wMinMax =
  let (sMinMax,aMinMax) = stepOne vMinMax wMinMax
  in (stepTwo sMinMax aMinMax, calcBuffer sMinMax aMinMax)
```

Figure 4: Self-defined types, imports and program entry. Left - Mathematical definitions. Right - *Haskell* implementation.

The right side of figure 4 shows the first part of the source code. The code begins with a *GHC* compiler flag for turning on the *OverloadedLists* extension. *OverloadedLists* allows to treat other container data structures syntactically as lists [8]. The implementation utilizes this to improve the readability for functions that use the imported array type `Vector`. Further, the figure shows that the implementation abstracts the types $\mathbb{R}$ and $\mathbb{N}$ to the data types `Float` and `Int`, respectively. Lastly, the main function of the safety zone computation `safetyZone` is shown. It takes the bounds for the translational and rotational velocity as an input, applies the input to the first step of the algorithm, applies the resulting output to the second step and the calculation for the buffer radius q, which then yields the final result.

$M = (v_0, s_0), \ldots, (v_m, s_m)$

$\Delta t = \ldots$

$\mathfrak{R} = R_1, \ldots, R_n$

$L = \ldots$

$D = \max_i |R_i|$

```
brakingMeasurements :: Vector (Velocity, Distance)
brakingMeasurements = [(0,0), (203.86,113.5),
      (535.131,365)]

latency :: Time
latency = 0.06

robotPoints :: [Point]
robotPoints = [(−233.5,−162.5), (193.5,−162.5),
      (193.5,162.5), (−233.5,162.5)]

l :: Float
l = fromIntegral (8 :: Int)

d :: Float
d = maximum (map vLen robotPoints)
  where
    vLen (x,y) = sqrt (x*x + y*y)
```

Figure 5: Configurable variables with a specific configuration in the *Haskell* implementation.

Figure 5 shows the mathematical definition and the actual implementation of the configuration variables. Despite L being a natural number, its counterpart `l` is of type `Float`. Since `l` is only used in floating point calculations, frequent explicit conversions from `Int` to `Float` would be necessary, due to *Haskell*'s strict type system. This would hinder readability and could affect performance. Considering that l still needs to be an integer, an explicit type annotation and conversion were added to the definition.

The actual implementation of the first step of the algorithm is depicted in figure 6 with `stepOne` as its main function. The function corresponds to the final result of the first step (s_{min}, s_{max}, $\alpha_{min}, \alpha_{max}$) with the components defined locally within a `where` clause. Further, the figure shows the greatest divergence between the implementation and the mathematical specification, since the index j has to be found algorithmically. The implementation does this with a variation of a binary search in the function `findIndex`. This is also the reason for the usage of the `Vector` type as a storage of the braking measurements, since it allows to access elements in constant time.

Figure 7 displays the required matrix operations for the second step of the algorithm. Only an ad hoc solution specialized to the necessary operations in step two was implemented. The operations rely heavily on pattern matching. Additionally, the matrix multiplication operation <...> factors in the fact that the $\mathbb{R}^2$ vector $\binom{x}{y}$ is implicitly extended to its *homogeneous coordinate* $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$.

The implementation of step two is illustrated in figure 8. The multiset that represents the convex hull is captured by the main function of the second step `stepTwo`. Each point definition is translated into a function, in which the subscript is lifted to a proper argument of said function, e.g.

$V \times \Omega$ with

$$V = \begin{cases} \{v_{min}, 0, v_{max}\} & 0 \in [v_{min}, v_{max}] \\ \{v_{min}, v_{max}\} & 0 \notin [v_{min}, v_{max}] \end{cases}$$

$$\Omega = \begin{cases} \{\omega_{min}, 0, \omega_{max}\} & 0 \in [\omega_{min}, \omega_{max}] \\ \{\omega_{min}, \omega_{max}\} & 0 \notin [\omega_{min}, \omega_{max}] \end{cases}$$

```
mkCandidates :: (Float, Float) → (Float, Float) → [(
    Float,Float)]
mkCandidates (vMin, vMax) (wMin, wMax) = [(v,w) | v ←
    velos, w ← omega] where
  velos
    | 0 ≥ vMin && 0 ≤ vMax = [vMin,0,vMax]
    | otherwise = [vMin,vMax]
  omega
    | 0 ≥ wMin && 0 ≤ wMax = [wMin,0,wMax]
    | otherwise = [wMin,wMax]
```

$$BM\Delta t(v, \omega) = \left(\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s} + \Delta t \right) \begin{pmatrix} v \\ \omega \end{pmatrix}$$

```
bm :: (Float, Float) → (Float, Float)
bm (v,w) = let ts' = ts (v,w) in ((ts'+latency)*v,(ts'+
    latency)*w)
```

$\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s}$ with

$$\hat{v}_s = \sqrt{v^2 + D^2\omega^2}$$

$$\hat{s}(v) = \begin{cases} \frac{s_1}{v_1} v & \text{if } v \leq v_1 \\ \frac{s_m}{v_m^3} v^3 & \text{if } v \geq v_m \\ s_{j-1} + \frac{s_j - s_{j-1}}{v_j - v_{j-1}}(v - v_{j-1}) & \text{otherwise} \end{cases}$$

```
ts :: (Float, Float) → Float
ts (v,w)
  | vs ≥ vm = sm / (vm*vm*vm) * vs * vs
  | vs ≤ v1 = s1 / v1
  | otherwise =
    let
      j' = findIndex vs
      (vj',sj') = brakingMeasurements ! j'
      (vj,sj) = brakingMeasurements ! (j'+1)
    in
      (sj' + ((sj-sj')/(vj-vj'))) * (vs-vj')) / vs
    where
      vs = sqrt (v*v+(d*d)*(w*w))
      (vm,sm) = brakingMeasurements ! (length
          brakingMeasurements - 1)
      (v1,s1) = brakingMeasurements ! 1
```

j such that $v_{j-1} \leq v \leq v_j$

```
findIndex :: Velocity → Int
findIndex v = go 0 (length brakingMeasurements-1) where
  go imin imax = let i = (imax+imin) `div` 2 in
    if imax-imin > 1
    then if fst (brakingMeasurements ! i) < v
         then go i imax
         else go imin i
    else imin
```

$(s_{min}, s_{max}, \alpha_{min}, \alpha_{max})$ with

$s_{min} = \min_c s^c$

$s_{max} = \max_c s^c$

$\alpha_{min} = \min_c \alpha^c$

$\alpha_{max} = \max_c \alpha^c$

and

$(s^c, \alpha^c) \in \left\{ BM\Delta t(v^c, w^c) \mid (v^c, w^c) \in V \times \Omega \right\}$

```
stepOne :: (Float, Float) → (Float, Float) → ((Float,
    Float),(Float, Float))
stepOne (vMin, vMax) (wMin, wMax) = ((sMin,sMax), (aMin,
    aMax)) where
  candidates = mkCandidates (vMin, vMax) (wMin, wMax)
  (sCandidates, αCandidates) = unzip (map bm candidates)
  sMin = minimum sCandidates
  sMax = maximum sCandidates
  aMin = minimum αCandidates
  aMax = maximum αCandidates
```

Figure 6: Step one of the algorithm.

$P^1_{i,s,\alpha} = R_i$ becomes $p1 = \lambda R_i \to \lambda(s, \alpha) \to R_i$. `stepTwo` constructs the final result by applying all combinations of elements of I and $\mathfrak{R}$ to these functions.

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \end{pmatrix}$$

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix}$$

$$s \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} sx \\ sy \end{pmatrix}$$

$$\begin{pmatrix} a_0 & a_1 \\ b_0 & b_1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_0 x + a_1 y \\ b_0 x + b_1 y \end{pmatrix}$$

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_0 x + a_1 y + a_2 \\ b_0 x + b_1 y + b_2 \end{pmatrix}$$

```
infixl 6 <->
(<->) :: Num a => (a,a) -> (a,a) -> (a,a)
(x1,y1) <-> (x2,y2) = (x1-x2,y1-y2)

infixl 6 <+>
(<+>) :: Num a => (a,a) -> (a,a) -> (a,a)
(x1,y1) <+> (x2,y2) = (x1+x2,y1+y2)

infixl 7 <.>
(<.>) :: Num a => a -> (a,a) -> (a,a)
s <.> (x,y) = (s*x,s*y)

infixl 7 <..>
(<..>) :: ((Float, Float), (Float, Float)) -> Point ->
      Point
((a0,a1),(b0,b1)) <..> (x,y) = (a0*x + a1*y, b0*x + b1*y)

infix 7 <.>
(<.>) :: ((Float, Float, Float), (Float, Float, Float))
      -> Point -> Point
((a0,a1,a2),(b0,b1,b2)) <.> (x,y) =
  (a0*x+a1*y+a2, b0*x+b1*y+b2)
```

Figure 7: Matrix operators used in step two of the algorithm.

As mentioned before, the implementation aims to be very closely related to the specification. This means that the implementation does not fully optimize the required definitions. For example, the final result $[[[P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]_{j=0}^{L-1}]_{i=0}^{n}]_{s_{min}}^{s_{max}}]_{\alpha_{min}}^{\alpha_{max}}$ contains each of the vehicle contour points four times, since $P^1_{i,s,\alpha}$ is defined independently of s and α but the final result contains $P^1_{i,s,\alpha}$ for all $(s, \alpha) \in I$. Therefore, the implementation mimics this behaviour.
The calculation of the buffer radius q is summarized in the function `calcBuffer`.

$$\operatorname{sinc}\phi = \begin{cases} \frac{\sin\phi}{\phi} & \phi \neq 0 \\ 1 & \phi = 0 \end{cases}$$

```
sinc :: Float → Float
sinc o
  | o ≠ 0 = sin o / o
  | otherwise = 1
```

$$T(s,\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & s \cdot \operatorname{sinc}\frac{\alpha}{2} \cdot \cos\frac{\alpha}{2} \\ \sin\alpha & \cos\alpha & s \cdot \operatorname{sinc}\frac{\alpha}{2} \cdot \sin\frac{\alpha}{2} \end{pmatrix}$$

```
tMatrix :: Float → Float →
  ((Float, Float, Float), (Float, Float, Float))
tMatrix s a = ((a0,a1,a2),(b0,b1,b2)) where
  a0 = cos a
  a1 = −sin a
  a2 = s ∗ sinc (a/2) ∗ cos (a/2)
  b0 = sin a
  b1 = cos a
  b2 = s ∗ sinc (a/2) ∗ sin (a/2)
```

$$Q(\alpha) = \begin{pmatrix} 1 & \tan\frac{\alpha}{2} \\ -\tan\frac{\alpha}{2} & 1 \end{pmatrix}$$

```
qMatrix :: Float → ((Float, Float), (Float, Float))
qMatrix a = ((1, tan (a/2)), (−tan (a/2), 1))
```

$$I = \{s_{min}, s_{max}\} \times \{\alpha_{min}, \alpha_{max}\}$$

```
mkInterval :: (Float, Float) → (Float, Float) → [(Float
    , Float)]
mkInterval (smin,smax) (amin,amax) =
  [ (s,a) | s ← [smin,smax], a ← [amin,amax] ]
```

$$P^1_{i,s,\alpha} = R_i$$

```
p1 :: Point → (Float, Float) → Point
p1 = const
```

$$P^2_{i,s,\alpha} = T(s,\alpha)R_i$$

```
p2 :: Point → (Float, Float) → Point
p2 ri (s,a) = tMatrix s a <.> ri
```

$$U^2_{i,s,\alpha} = T\left(\frac{s}{L}, \frac{\alpha}{L}\right) \cdot R_i$$

```
u2 :: Point → (Float, Float) → Point
u2 ri (s,a) = tMatrix (s/l) (a/l) <.> ri
```

$$U^3_{i,s,\alpha} = U^1_{i,s,\alpha} + Q(\frac{\alpha}{L})\frac{1}{2}(U^2_{i,s,\alpha} - U^1_{i,s,\alpha})$$

```
u3 :: Point → (Float, Float) → Point
u3 u1@ri sa@(s,a) = u1 <+> qMatrix (a/l) <..> (0.5 <.> (
    u2 ri sa <-> u1))
```

$$V^j_{i,s,\alpha} = T\left(\frac{j \cdot s}{L}, \frac{j \cdot \alpha}{L}\right) \cdot U^3_{i,s,\alpha}$$

```
v :: Float → Point → (Float, Float) -> Point
v j ri sa@(s,a) = tMatrix ((j∗s)/l) ((j∗a)/l) <.> u3 ri
    sa
```

$$[[[P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]^{L-1}_{j=0}]^n_{i=0}]^{s_{max}}_{s_{min}}]^{\alpha_{max}}_{\alpha_{min}}$$

```
stepTwo :: (Float,Float) → (Float, Float) → [Point]
stepTwo (smin,smax) (amin,amax) =
  [ f ri sa | sa ← interval, ri ← robotPoints, f ←
      p1p2v ]
  where
    interval = mkInterval (smin,smax) (amin,amax)
    p1p2v = p1 : p2 : map v [0..l−1]
```

$$q = q^A + q^B$$

$$q^A = \frac{1}{6}\left(\frac{\alpha_{max} - \alpha_{min}}{2}\right)^2 \max\{|s_{min}|; |s_{max}|\}$$

$$q^B = \left(1 - \cos\frac{\alpha_{max} - \alpha_{min}}{2}\right) \max_i |R_i|$$

```
calcBuffer :: (Float, Float) → (Float, Float) → Float
calcBuffer (sMin, sMax) (aMin, aMax) =
  1/6 ∗ (av∗av) ∗ maxS + (1−cos av) ∗ maxR
  where
    av = (aMax−aMin)/2
    maxS = max (abs sMin) (abs sMax)
    maxR = d
```

Figure 8: Step two of the algorithm.

3.2 Verification

This section will provide the actual verification of the previously presented implementation of the algorithm. Unfortunately, the verification process was not as straightforward as expected. Hence, this chapter will also provide insights into the verification process.

3.2.1 First Attempt in *Isabelle*

As a first attempt it was decided to try to verify the implementation with the aid of a theorem prover, since machine-assisted proofs are less likely to contain logical errors [1, p. 142-144]. The choice for a theorem prover fell on *Isabelle*, because the original case study was verified in it as well. *Isabelle* is an interactive theorem prover. Proofs written in *Isabelle* can use *Higher Order Logic* (*HOL*), which is described as the composition of functional programming and logic, to prove certain properties [15, p. 3]. The fact that *HOL* can be used as a functional programming language was utilized to easily translate the *Haskell* implementation to *Isabelle/HOL*. Figure 9 shows the translated implementation of the `ts` function as an example. Since *Isabelle* does not provide *syntactic sugar* like *guards* or *where clauses*, the translation uses *if-expressions* and *let-bindings*, respectively. The types `Float` and `Int` were translated to the types `real` and `nat`, which in *Isabelle* denote $\mathbb{R}$ and $\mathbb{N}$.

```
fun ts :: "real * real ⇒ real" where
"ts (v,w) =
  (let
    vs = sqrt (v*v+d*d*w*w);
    (vm,sm) = brakingMeasurements ! (length brakingMeasurements − 1);
    (v1,s1) = brakingMeasurements ! 1
  in
    if vs ≥ vm then sm / (vm*vm*vm) * vs * vs
    else
      if vs ≤ v1 then s1 / v1
      else
        (let
          j' = findIndex vs;
          (vj', sj') = brakingMeasurements ! j';
          (vj, sj ) = brakingMeasurements ! (j'+1)
        in (sj' + ((sj−sj')/(vj−vj')) * (vs−vj')) / vs))"
```

Figure 9: The implementation of the `ts` function in *Isabelle*.

Functions that contain *list comprehensions* as well as *arithmetic sequences* were first translated to their kernel according to the *Haskell Language Report* [11, p. 21-22], before they were translated further. Figure 10 shows an example of the translated version of `mkInterval`. Additionally,

functions that were not included in the *Isabelle Main* library, e.g. `concatMap`, were implemented accordingly to the *Haskell Prelude* module [11, p. 105-122]. In one case, i.e. `enumFromTo`, an additional termination proof was needed, because it could not be derived automatically by *Isabelle*.

```
fun mkInterval :: "real * real ⇒ real * real ⇒
    (real * real) list" where
"mkInterval (smin,smax) (amin,amax) =
  concatMap (λs. concatMap (λa. [(s,a)]) [amin,amax]) [smin,smax]"
```

Figure 10: The implementation of the `mkInterval` function in *Isabelle*.

Moreover, the verification process was initiated by implementing a formal description of the central safety guarantee of the algorithm. In the original paper this guarantee is described semi-formally: *"The algorithm guarantees that for any velocity* $v \in [v_{min}, v_{max}]$ *and any rotational velocity* $\omega \in [\omega_{min}, \omega_{max}]$ *no part of the vehicle will leave the safety zone at any time while first driving with constant velocity* $(v, \omega)^T$ *for time* Δt *and then braking down to standstill according to the braking model."* [20, p. 3]. This guarantee was further formalized by using set theory notation. The resulting main theorem and the depending definitions are shown in figure 12.

Unfortunately, this attempt was not very successful. *Isabelle* reported 18 unproven sub-theorems after instructing it to solve the theorem automatically. Due to the inexperience of the author with *Isabelle* and time constraints of this thesis this approach was not pursued any further. However, *Isabelle* proved to be very useful over the course of the remaining verification process, since it was possible to prove more general properties in *Isabelle* with ease. These properties were used to reason about the implementation without the tedious act of proving them explicitly. For instance, the property concatMap $f\ xs =$ concat (map $f\ xs$) was easily accepted by *Isabelle* just by instructing it to perform an induction proof over the list variable xs as seen in Figure 11.

```
lemma concatMapSimplified [simp]: "concatMap f xs = concat (map f xs)"
  apply (induction xs)
  apply auto
done
```

Figure 11: Example of automatic proof performed by *Isabelle*.

```
definition segment :: "(real * real) ⇒ (real * real) ⇒ Point set" where
  "segment u v' = { z. ∃α ∈ ℝ. 0 ≤ α ∧ α ≤ 1 ∧ (z = (α <.> u <+> (1−α) <.> v'))
      }"

definition is_convex :: "Point set ⇒ bool" where
  "is_convex K = (∀u ∈ K. ∀v ∈ K. segment u v ⊆ K)"

definition conv :: "Point set ⇒ Point set" where
  "conv X = ∩{K. is_convex K ∧ X ⊆ K}"

definition straightBrakingDistance :: "real ⇒ real" where
"straightBrakingDistance v' =
  (let
    j   = The (λx. (fst (brakingMeasurements ! (x−1))) ≤ v'
                 ∧ v' ≤ (fst (brakingMeasurements ! x)));
    (vj',sj') = brakingMeasurements ! (j−1);
    (vj,sj) = brakingMeasurements ! j;
    (v1,s1) = brakingMeasurements ! 1;
    (vm,sm) = brakingMeasurements ! (length brakingMeasurements − 1)
  in
    (if v' ≤ v1 then s1/v1*v'
     else if v' ≥ vm then sm/(vm*vm*vm)*v'*v'*v'
     else  sj' + (sj−sj') / (vj−vj') * (v'−vj')))"

fun BMΔt :: "real * real ⇒ real * real" where
  "BMΔt (v',ω) =
  (let
    vsHat = sqrt (v'^2+d^2*ω^2);
    sHat = straightBrakingDistance vsHat
  in
    (sHat / vsHat + latency) <.> (v',ω) )"

definition ℌ :: "real ⇒ real ⇒ (real * real) set" where
  "ℌ s α = Union{ {tMatrix (x*s) (x*α) <.> r
  | r. r ∈ (set robotPoints)} | x. x ∈ ℝ ∧ x ≥ 0 ∧ x ≤ 1 }"

theorem mainSafetyTheorem :
  "(v',ω) ∈ ℝ × ℝ ∧ v' ≥ vMin ∧ v' ≤ vMax ∧ ω ≥ ω ∧ ω ≤ ω →
    (let
      ℋ = conv (set (fst (safetyZone (vMin,vMax) (ω, ω))));
      (s,α) = BMΔt(v',ω)
     in
      ℋ ⊇ (ℌ s α))"
  apply clarify
  apply (unfold Let_def)
  apply auto
  sorry
```

Figure 12: The central safety guarantee formalized in *Isabelle*.

Since the machine-aided attempt was not successful, the next approach was to try to verify the implementation without the help of a computer by just reasoning about the code. This approach turned out to be successful. Thus, the rest of this chapter will present this formal reasoning in detail.

3.2.2 Verification by Formal Reasoning

Several assumptions were made beforehand. First, it is assumed that `Double` and `Int` are correctly represent $\mathbb{R}$ and $\mathbb{N}$, respectively, which was also an assumption that the original paper made [20, p. 5]. Further, it is assumed that the basic arithmetic operations on $\mathbb{R}$ and $\mathbb{N}$, namely *addition*, *subtraction*, *multiplication* and *division*, are correctly represented by the *Haskell* functions `(+)`, `(-)`, `(*)` and `(/)`.
Additionally, it is assumed that the used functions from the `Prelude` and the `Vector` packages work correctly as well. It is also assumed that the input of the algorithm as well as the configurations variables are appropriate and that they are equivalent to their mathematical counterpart. At last, in order to prove *total correctness* of an algorithm, the algorithm needs to be *partially correct* as well as *terminating*. In the following proofs, it is assumed that *partial correctness* implies *total correctness* as well. This assumption is based on the fact that the implementation is only using finite instances of data structures and implicit recursion through library higher-order functions, e.g. map, which are assumed to be *totally correct* for finite inputs[1].
Furthermore, in each of the following proofs it is necessary to show that the implementation is equivalent to the specification. Since the specification describes its definitions in the terms of multisets whereas the implementation uses more efficient data structures like lists or arrays, proofs of equivalence between terms of the implementation and the specification would in general not be possible. Hence, the proofs are restricted to equivalence up to isomorphism. Informally a specification term t_{spec} is equivalent up to isomorphism to its implementation term t_{impl} (written: $t_{spec} \cong t_{impl}$) if t_{spec} would be equal to t_{impl} if the type of structure for both terms is neglected. Thus, t_{spec} is assumed to be equal to t_{impl} if both are isomorphic.

The following proof will refer to the *Haskell* implementation shown in chapter 3.1 in the figures 4-8. Despite that, for ease of readability, the rest of the section will repeat definitions from the specification (chapter 2.2) and the implementation (chapter 3.1) if necessary.

The proof starts by looking at the entry function safetyZone. As one can see, the input of the function is passed to stepOne, which yields the result of the first step (sMinMax,wMinMax). The result is then the input of the main function of the second step of the algorithm stepTwo and the buffer calculation calcBuffer. The composition of both function applications yields the final

[1] The function findIndex is an exception to that assumption, since it introduces explicit recursion.

result. Hence, in order to prove that the final result is correct, it first needs to be shown that the result of stepOne is correct.

```
safetyZone :: (Float, Float) → (Float, Float) → SafetyZone
safetyZone vMinMax wMinMax =
  let (sMinMax,aMinMax) = stepOne vMinMax wMinMax
  in (stepTwo sMinMax aMinMax, calcBuffer sMinMax aMinMax)
```

Figure 13: Entry function of the implementation.

The definition of stepOne is shown in figure 14. The results of the function are the extremes of the

$(s_{min}, s_{max}, \alpha_{min}, \alpha_{max})$ with

$$s_{min} = \min_c s^c$$

$$s_{max} = \max_c s^c$$

$$\alpha_{min} = \min_c \alpha^c$$

$$\alpha_{max} = \max_c \alpha^c$$

and

$$(s^c, \alpha^c) \in \left\{ BM\Delta t(v^c, w^c) \mid (v^c, w^c) \in V \times \Omega \right\}$$

```
stepOne :: (Float, Float) → (Float, Float) → ((Float,
      Float),(Float, Float))
stepOne (vMin, vMax) (wMin, wMax) = ((sMin,sMax), (aMin,
      aMax)) where
  candidates = mkCandidates (vMin, vMax) (wMin, wMax)
  (sCandidates, αCandidates) = unzip (map bm candidates)
  sMin = minimum sCandidates
  sMax = maximum sCandidates
  aMin = minimum αCandidates
  aMax = maximum αCandidates
```

Figure 14: Entry function of the first step.

sCandidates and the αCandidates lists. These lists are themselves results of the computation of map bm, which takes the candidates list as an input. candidates is defined in terms of mkCandidates (see figure 15 right).

$V \times \Omega$ with

$$V = \begin{cases} \{v_{min}, 0, v_{max}\} & 0 \in [v_{min}, v_{max}] \\ \{v_{min}, v_{max}\} & 0 \notin [v_{min}, v_{max}] \end{cases}$$

$$\Omega = \begin{cases} \{\omega_{min}, 0, \omega_{max}\} & 0 \in [\omega_{min}, \omega_{max}] \\ \{\omega_{min}, \omega_{max}\} & 0 \notin [\omega_{min}, \omega_{max}] \end{cases}$$

```
mkCandidates :: (Float, Float) → (Float, Float) → [(
      Float,Float)]
mkCandidates (vMin, vMax) (wMin, wMax) =
  [(v,w) | v ← velos, w ← omega] where
  velos
    | 0 ≥ vMin && 0 ≤ vMax = [vMin,0,vMax]
    | otherwise = [vMin,vMax]
  omega
    | 0 ≥ wMin && 0 ≤ wMax = [wMin,0,wMax]
    | otherwise = [wMin,wMax]
```

Figure 15: Definition of the candidates.

It was shown via *Isabelle's* simplification mechanism how the candidates are constructed based on their input[2]. The exhausting construction of the candidates are shown in the equations (25)-(28).

$$0 \in [vMin, vMax] \wedge 0 \in [\omega Min, \omega Max] \Rightarrow$$
$$\text{mkCandidates } (vMin, vMax)\ (\omega Min, \omega Max) = [(vMin, \omega Min), (vMin, 0),$$
$$(vMin, \omega Max), (0, \omega Min), (0, 0), (0, \omega Max), (vMax, \omega Min), (vMax, 0), (vMax, \omega Max)] \quad (25)$$

$$0 \notin [vMin, vMax] \wedge 0 \in [\omega Min, \omega Max] \Rightarrow$$
$$\text{mkCandidates } (vMin, vMax)\ (\omega Min, \omega Max) = [(vMin, \omega Min), (vMin, 0),$$
$$(vMin, \omega Max), (vMax, \omega Min), (vMax, 0), (vMax, \omega Max)] \quad (26)$$

$$0 \in [vMin, vMax] \wedge 0 \notin [\omega Min, \omega Max] \Rightarrow$$
$$\text{mkCandidates } (vMin, vMax)\ (\omega Min, \omega Max) = [(vMin, \omega Min), (vMin, \omega Max),$$
$$(0, \omega Min), (0, \omega Max), (vMax, \omega Min), (vMax, \omega Max)] \quad (27)$$

$$0 \notin [vMin, vMax] \wedge 0 \notin [\omega Min, \omega Max] \Rightarrow$$
$$\text{mkCandidates } (vMin, vMax)\ (\omega Min, \omega Max) =$$
$$[(vMin, \omega Min), (vMin, \omega Max), (vMax, \omega Min), (vMax, \omega Max)] \quad (28)$$

This thesis refrains from unfolding $V \times \Omega$ (see figure 15 left) based on $(v_{min}, v_{max}, \omega_{min}, \omega_{max})$, but a trained reader can easily see, that (29) holds by considering (25)-(28).

$$\mathsf{mkCandidates\ \ (vMin, vMax)\ \ (wMin, wMax)} \cong V \times \Omega \quad (29)$$

□

`candidates` is applied to `map bm`. Hence, every element of `candidates` is applied to `bm`, which is why `bm` needs to be inspected next.

[2] The lemmas can be found in Appendix B in the *Isabelle* theory code under the names `mkCandidatesResult_1` - `mkCandidatesResult_4`.

$$BM\Delta t(v,\omega) = \left(\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s} + \Delta t\right)\begin{pmatrix} v \\ \omega \end{pmatrix}$$

```
bm :: (Float, Float) → (Float, Float)
bm (v,w) = let ts' = ts (v,w) in ((ts'+latency)*v,(ts'+
    latency)*w)
```

$$\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s} \quad \text{with}$$

$$\hat{v}_s = \sqrt{v^2 + D^2\omega^2}$$

$$\hat{s}(v) = \begin{cases} \frac{s_1}{v_1}v & \text{if } v \leq v_1 \\ \frac{s_m}{v_m^3}v^3 & \text{if } v \geq v_m \\ s_{j-1} + \frac{s_j - s_{j-1}}{v_j - v_{j-1}}(v - v_{j-1}) & \text{otherwise} \end{cases}$$

```
ts :: (Float, Float) → Float
ts (v,w)
  | vs ≥ vm = sm / (vm*vm*vm) * vs * vs
  | vs ≤ v1 = s1 / v1
  | otherwise =
    let
      j' = findIndex vs
      (vj',sj') = brakingMeasurements ! j'
      (vj,sj) = brakingMeasurements ! (j'+1)
    in
      (sj' + ((sj-sj')/(vj-vj')) * (vs-vj')) / vs
    where
      vs = sqrt (v*v+(d*d)*(w*w))
      (vm,sm) = brakingMeasurements ! (length
          brakingMeasurements - 1)
      (v1,s1) = brakingMeasurements ! 1
```

j such that $v_{j-1} \leq v \leq v_j$

```
findIndex :: Velocity → Int
findIndex v = go 0 (length brakingMeasurements-1) where
  go imin imax = let i = (imax+imin) `div` 2 in
    if imax-imin > 1
    then if fst (brakingMeasurements ! i) < v
         then go i imax
         else go imin i
    else imin
```

Figure 16: Definition of `bm`.

One can see by looking at figure 16 that `bm (v,w)` would be isomorphic to $BM\Delta t(v,w)$ if `ts'` is isomorphic to $\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s}$. Thus, this needs to be shown first. Since `ts'` is the result of `ts (v,w)`, `ts` needs to be inspected. `ts` locally defines three variables in its where-clause (see figure 16). The first one `vs` would be equivalent to the equivalent straight velocity $\hat{v}_s$, if `d` is equivalent to D.

$$D = \max_i |R_i|$$

```
d :: Float
d = maximum (map vLen robotPoints)
  where
    vLen (x,y) = sqrt (x*x + y*y)
```

Figure 17: Definition of `d`.

The definitions of D and `d` are displayed in figure 17. From (30) one arrives at (32), by applying *equational reasoning*.

$$
\begin{aligned}
d &= \text{maximum (map } vLen\ robotPoints) && (30)\\
&\quad \texttt{where } vLen(x,y) = \text{sqrt } (x*x\ +\ y*y)\\
&= \text{maximum (map } (\lambda(x,y) \to \text{sqrt } (x*x\ +\ y*y))\ robotPoints) && (31)\\
&= \text{maximum } [\text{sqrt}(x_1*x_1\ +\ y_1*y_1),\ \ldots\ ,\text{sqrt}(x_n*x_n\ +\ y_n*y_n)] && (32)
\end{aligned}
$$

Since it is assumed that `maximum` and `sqrt` work correctly as well as $\mathfrak{R}$ being isomorphic to `robotPoints`, `vs` $\cong \hat{v}_s$ follows. `(v1,s1)` $\cong (v_1, s_1)$ and `(vm,sm)` $\cong (v_m, s_m)$ follows, because of the assumptions that `brakingMeasurements` is isomorphic to M and that the library functions, i.e. `(!)` and `length`, work correctly. Next, `ts (v,w)` $\cong \frac{\hat{s}(\hat{v}_s)}{\hat{v}_s}$ needs to be shown. Because `ts (v,w)` as well as $\hat{s}(\hat{v}_s)$ are piecewise functions sharing the same conditions, both directions ("$\Rightarrow$" and "$\Leftarrow$") of the equivalence proof will be dealt with at the same time.

Case 1: $\hat{v}_s \geq v_m$*:*

(33) shows the result of $\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s}$ for this case.

$$
\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s} \ = \ \frac{\frac{s_m}{v_m^3}\hat{v}_s^3}{\hat{v}_s} \ = \ \frac{s_m}{v_m^3}\hat{v}_s^2 \ = \ s_m \ / \ (v_m \cdot v_m \cdot v_m) \cdot \hat{v}_s \cdot \hat{v}_s \qquad (33)
$$

`ts (v,w)` evaluates to `sm / (vm*vm*vm) * vs * vs`, which is equivalent to (33).

Case 2: $\hat{v}_s \leq v_1$*:*

(34) shows the result of $\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s}$ for this case.

$$
\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s} \ = \ \frac{\frac{s_1}{v_1}\hat{v}_s}{\hat{v}_s} \ = \ \frac{s_1}{v_1} \qquad (34)
$$

`ts (v,w)` evaluates to `s1 / v1`, which is equivalent to (34).

Case 3: otherwise ($\hat{v}_s < v_m \wedge \hat{v}_s > v_1$)*:*

(35) shows the result of $\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s}$ for this case.

$$
\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s} \ = \ \frac{s_{j-1} + \frac{s_j - s_{j-1}}{v_j - v_{j-1}}(\hat{v}_s - v_{j-1})}{\hat{v}_s} \quad \text{with } j \text{ such that } v_{j-1} \leq \hat{v}_s \leq v_j \qquad (35)
$$

`ts (v,w)` evaluates to (36), which would be isomorphic to $\frac{\hat{s}(\hat{v}_s)}{\hat{v}_s}$ if `j'` would satisfy the same property that $j-1$ from (35) does.

$$
\begin{aligned}
ts\ (v,\omega) = \texttt{let}\ & \\
& j' = \text{findIndex}\ vs \\
& (vj', sj') = brakingMeasurements\ !\ j' \\
& (vj, sj) = brakingMeasurements\ !\ (j'+1) \\
\texttt{in}\ & (sj' + ((sj - sj')/(vj - vj')) * (vs - vj')) \ /\ vs
\end{aligned} \tag{36}
$$

Hence, it needs to be shown that `findIndex v` results in an index `j'` such that $v_{j'} \leq v \leq v_{j'+1}$. This will be done by strong induction over the difference of the arguments of the locally defined `go` function. The definition of `findIndex` (and therefore `go`) is displayed in figure 16. In the following proof it will again be assumed that `brakingMeasurements` refers to M. By definition of M, `breakingMeasurements` is an ascending order and contains at least 2 elements[3]. Because $\hat{v}_s < v_m \wedge \hat{v}_s > v_1$ and therefore `vs` $<$ `vm` $\wedge$ `vs` $>$ `v1`, one can assume that there exists always an index with the desired property within the range of `0` and `length brakingMeasurements` -1.

<u>*Base case*</u>: $imax - imin = 0$. go $imin\ imax\ =\ imin$. Because of the assumptions, `findIndex v` works correctly for this case.

<u>*Induction Hypothesis*</u>: Assume that `findIndex v` works correctly, i.e. it results in an index j' such that $v_{j'} \leq v \leq v_{j'+1}$ for $imax - imin \leq n$.

<u>*Induction Step*</u>: $imax - imin\ =\ n+1$.

Case 1: fst $(brakingMeasurements\ !\ i) < v$

$$
\begin{aligned}
\text{go}\ imin\ imax = \texttt{let}\ i\ =\ & (imax + imin)\ `div`\ 2\ \texttt{in}\ \text{go}\ i\ imax && (37)\\
=\ & \text{go}\ ((imax + imin)\ `div`\ 2)\ imax && (38)
\end{aligned}
$$

Case 1.1: $imax + imin$ is even.
One can define the new difference of the arguments of the next `go` application k.

$$k = imax - i = imax - ((imax + imin)\ /\ 2) \tag{39}$$

[3] Actually, `brakingMeasurements` needs to contain at least 3 elements, since $\hat{v}_s < v_m \wedge \hat{v}_s > v_1 \Rightarrow v_1 \neq v_m$.

By definition of `ts` one can assume $imax - imin \geq 2$ and therefore $imax \geq imin + 2$, which is used in the following to show that $i > imin$.

$$(imax + imin) \ / \ 2 \ \geq \ (imin + imin + 2) \ / \ 2 \tag{40}$$
$$\Leftrightarrow (imax + imin) \ / \ 2 \ \geq \ imin + 1 \tag{41}$$
$$\Leftrightarrow i \ \geq \ imin + 1 \tag{42}$$
$$\Rightarrow i \ > \ imin \tag{43}$$

Because of (43) $k < n + 1$ follows and therefore the *IH* applies. *Case 1.2:* $imax + imin$ is odd. One can define the new difference of the arguments of the next `go` application k.

$$k = imax - i = imax - ((imax + imin - 1) \ / \ 2) \tag{44}$$

By definition of `ts` one can assume $imax - imin \geq 2$. Further, $imax - imin \geq 3$ follows, since $imax + imin$ is odd. Therefore, $imax \geq imin + 3$ follows, which is used in the following to show that $i > imin$.

$$(imax + imin - 1) \ / \ 2 \ \geq \ (imin + imin + 2) \ / \ 2 \tag{45}$$
$$\Leftrightarrow (imax + imin - 1) \ / \ 2 \ \geq \ imin + 1 \tag{46}$$
$$\Leftrightarrow i \ \geq \ imin + 1 \tag{47}$$
$$\Rightarrow i \ > \ imin \tag{48}$$

Because of (48) $k < n + 1$ follows and therefore the *IH* applies.

Case 2: fst $(brakingMeasurements \ ! \ i) \geq v$

$$\text{go} \ imin \ imax = \texttt{let} \ i \ = \ (imax + imin) \ `div` \ 2 \ \texttt{in} \ \text{go} \ imin \ i \tag{49}$$
$$= \ \text{go} \ imin \ ((imax + imin) \ `div` \ 2) \tag{50}$$

Since the rest of the case is symmetrical to the first case, it is omitted here for brevity.

By case 1 and 2 it is concluded that `findIndex v` results in an index j' such that $v_{j'} \leq v \leq v_{j'+1}$ and is therefore regarded as partially correct. Termination and thus total correctness is derived by the fact that the metric $imax - imin$ is decreasing with every application of `go`. Combined with the terminating base case $imax - imin \leq 1$, this yields termination[4].

□

[4] In addtion to this rather informal reasoning, there is a termination proof accepted by *Isabelle*, which can be found in Appendix B on page 56.

Therefore, (36) satisfies (35), since $(\mathsf{vj'}, \mathsf{sj'}) \cong (v_{j-1}, s_{j-1})$ and $(\mathsf{vj}, \mathsf{sj}) \cong (v_j, s_j)$. Further, by case 1-3 (51) is concluded.

$$\mathsf{ts}\ (\mathsf{v},\mathsf{w}) \cong \frac{\hat{s}(\hat{v}_s)}{\hat{v}_s} \tag{51}$$

□

With (51) one can conclude (52).

$$\mathsf{bm}\ (\mathsf{v},\mathsf{w}) \cong BM\Delta t(v, w) \tag{52}$$

Since map f [x1, ... ,xn] results in the list [f x1, ... , f xn] and (52) as well as (29), map bm candidates is isomorphic to the set of braking model candidates:

$$\mathsf{map\ bm\ candidates} \cong \left\{ BM\Delta t(v^c, w^c) \mid (v^c, w^c) \in V \times \Omega \right\} \tag{53}$$

As mentioned previously, *Isabelle* was used to show more general theorems. Two of those were the following two equations (54) and (55), which were proven by induction over xs and simplification, respectively.

$$\text{unzip}\ xs\ =\ (\text{map fst}\ xs,\ \text{map snd}\ xs) \tag{54}$$

$$\text{map}\ f\ (\text{map}\ g\ xs)\ =\ \text{map}\ (f\ .\ g)\ xs \tag{55}$$

Hence, (sCandidates, αCandidates) can be rewritten as in (57).

$$\begin{aligned} &(\mathsf{sCandidates}, \alpha\mathsf{Candidates}) \\ &= \mathsf{unzip}\ (\mathsf{map\ bm\ candidates}) && (56) \\ &= (\mathsf{map}\ (\mathsf{fst} \circ \mathsf{bm})\ \mathsf{candidates},\ \mathsf{map}\ (\mathsf{snd} \circ \mathsf{bm})\ \mathsf{candidates}) && (57) \end{aligned}$$

By considering (53) one arrives at (58) and (59).

$$\mathsf{sCandidates} \cong \{s_1, \ldots, s_b\} \tag{58}$$

$$\alpha\mathsf{Candidates} \cong \{\alpha_1, \ldots, \alpha_b\} \tag{59}$$

with $\left\{(s_1, \alpha_1), \ldots, (s_b, \alpha_b)\right\} = \left\{ BM\Delta t(v^c, w^c) \mid (v^c, w^c) \in V \times \Omega \right\}$. Since it is assumed that the library functions minimum and maximum work correctly and sCandidates as well as aCandidates are not empty, (sMin, sMax, aMin, aMax) refers to the extremes of the braking model. Thus, resulting

in (60)-(63).

$$\mathsf{sMin} = \mathsf{minimum}\ \mathsf{sCandidates} \ \cong\ \min_c s^c = s_{min} \tag{60}$$

$$\mathsf{sMax} = \mathsf{maximum}\ \mathsf{sCandidates} \ \cong\ \max_c s^c = s_{max} \tag{61}$$

$$\mathsf{aMin} = \mathsf{minimum}\ \alpha\mathsf{Candidates} \ \cong\ \min_c \alpha^c = \alpha_{min} \tag{62}$$

$$\mathsf{aMax} = \mathsf{maximum}\ \alpha\mathsf{Candidates} \ \cong\ \max_c \alpha^c = \alpha_{max} \tag{63}$$

Because $((\mathsf{sMin},\mathsf{sMax}),(\mathsf{aMin},\mathsf{aMax}))$ is also the result of $\mathsf{stepOne}\ \mathsf{vMinMax}\ \mathsf{wMinMax}$ (see figure 14), the verification of the first step of the algorithm is finally concluded by (64).

$$\mathsf{stepOne}\ \mathsf{vMinMax}\ \mathsf{wMinMax} \cong (s_{min}, s_{max}, \alpha_{min}, \alpha_{max}) \tag{64}$$

□

The equation (64) states that stepOne satisfies the specification of the first step of the algorithm. Therefore, stepOne is regarded as correct. Hence, in the following it is assumed that the output of the function stepOne computes a correct result. Referring back to figure 13, the verification will proceed with the second step of the algorithm, namely stepTwo. Afterwards, the correctness of the buffer radius calculation will be shown.

$$\operatorname{sinc}\phi = \begin{cases} \frac{\sin\phi}{\phi} & \phi \neq 0 \\ 1 & \phi = 0 \end{cases}$$

```
sinc :: Float → Float
sinc o
  | o ≠ 0 = sin o / o
  | otherwise = 1
```

$$T(s,\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & s \cdot \operatorname{sinc}\frac{\alpha}{2} \cdot \cos\frac{\alpha}{2} \\ \sin\alpha & \cos\alpha & s \cdot \operatorname{sinc}\frac{\alpha}{2} \cdot \sin\frac{\alpha}{2} \end{pmatrix}$$

```
tMatrix :: Float → Float →
  ((Float, Float, Float), (Float, Float, Float))
tMatrix s a = ((a0,a1,a2),(b0,b1,b2)) where
  a0 = cos a
  a1 = −sin a
  a2 = s ∗ sinc (a/2) ∗ cos (a/2)
  b0 = sin a
  b1 = cos a
  b2 = s ∗ sinc (a/2) ∗ sin (a/2)
```

$$Q(\alpha) = \begin{pmatrix} 1 & \tan\frac{\alpha}{2} \\ -\tan\frac{\alpha}{2} & 1 \end{pmatrix}$$

```
qMatrix :: Float → ((Float, Float), (Float, Float))
qMatrix a = ((1, tan (a/2)), (−tan (a/2), 1))
```

$$I = \{s_{min}, s_{max}\} \times \{\alpha_{min}, \alpha_{max}\}$$

```
mkInterval :: (Float, Float) → (Float, Float) → [(Float
    , Float)]
mkInterval (smin,smax) (amin,amax) =
  [ (s,a) | s ← [smin,smax], a ← [amin,amax] ]
```

$$P^1_{i,s,\alpha} = R_i$$

```
p1 :: Point → (Float, Float) → Point
p1 = const
```

$$P^2_{i,s,\alpha} = T(s,\alpha)R_i$$

```
p2 :: Point → (Float, Float) → Point
p2 ri (s,a) = tMatrix s a <.> ri
```

$$U^2_{i,s,\alpha} = T\left(\frac{s}{L}, \frac{\alpha}{L}\right) \cdot R_i$$

```
u2 :: Point → (Float, Float) → Point
u2 ri (s,a) = tMatrix (s/l) (a/l) <.> ri
```

$$U^3_{i,s,\alpha} = U^1_{i,s,\alpha} + Q(\frac{\alpha}{L})\frac{1}{2}(U^2_{i,s,\alpha} - U^1_{i,s,\alpha})$$

```
u3 :: Point → (Float, Float) → Point
u3 u1@ri sa@(s,a) = u1 <+> qMatrix (a/l) <..> (0.5 <.> (
    u2 ri sa <-> u1))
```

$$V^j_{i,s,\alpha} = T\left(\frac{j \cdot s}{L}, \frac{j \cdot \alpha}{L}\right) \cdot U^3_{i,s,\alpha}$$

```
v :: Float → Point → (Float, Float) -> Point
v j ri sa@(s,a) = tMatrix ((j∗s)/l) ((j∗a)/l) <.> u3 ri
    sa
```

$$[[[P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]^{L-1}_{j=0}]^n_{i=0}]^{s_{max}}_{s_{min}}]^{\alpha_{max}}_{\alpha_{min}}$$

```
stepTwo :: (Float,Float) → (Float, Float) → [Point]
stepTwo (smin,smax) (amin,amax) =
  [ f ri sa | sa ← interval, ri ← robotPoints, f ←
      p1p2v ]
  where
    interval = mkInterval (smin,smax) (amin,amax)
    p1p2v = p1 : p2 : map v [0..l−1]
```

Figure 18: Definition of `stepTwo`.

The definitions belonging to stepTwo are shown in figure 18, while the self defined matrix operation are shown in figure 7. The result of the second step of the algorithm is a multiset of points representing the convex hull $[[[P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]^{L-1}_{j=0}]^n_{i=0}]^{s_{max}}_{s_{min}}]^{\alpha_{max}}_{\alpha_{min}}$. It needs to be shown that stepTwo calculates the same points. This will be done via a bottom-up approach.

The proof starts by showing that $P^1_{i,s,\alpha}$ and $P^2_{i,s,\alpha}$ are isomorphic to p1 R_i (s,α) and p2 R_i (s,α),

respectively.

$P^1_{i,s,\alpha}$ is defined as R_i. Since p1 can be rewritten as $\texttt{p1} = \texttt{const} = \lambda R_i \to \lambda\_ \to R_i$, $\texttt{p1}\ R_i\ (s, \alpha)$ and $P^1_{i,s,\alpha}$ are isomorphic.

$$\texttt{p1}\ R_i\ (s, \alpha) \cong P^1_{i,s,\alpha} \tag{65}$$

□

The proof for the later assumption involves a little bit more work. First, just by looking at the definition, one can see that the definition of $\text{sinc}\,\phi$ and sinc o are almost identical. Hence, sinc $\cong$ sinc follows. Next, $\texttt{p2}\ R_i\ (s, \alpha)$ can be rewritten by applying *equational reasoning* as seen in (66)-(69) using the definitions of the figures 7 and 18.

$$\begin{aligned}
\texttt{p2}\ R_i\ (s, \alpha) &= \texttt{tMatrix}\ s\ \alpha\ \texttt{<...>}\ R_i && (66)\\
&= \texttt{let}\ (x_{R_i}, y_{R_i}) = R_i \\
&\quad \texttt{in tMatrix}\ s\ \alpha\ \texttt{<...>}\ (x_{R_i}, y_{R_i}) && (67)\\
&= \texttt{let}\ (x_{R_i}, y_{R_i}) = R_i \\
&\quad \texttt{in}\ ((a0, a1, a2), (b0, b1, b2))\ \texttt{<...>}\ (x_{R_i}, y_{R_i}) && (68)\\
&\quad\ \texttt{where} \\
&\quad\quad a0 = \cos\ \alpha \\
&\quad\quad a1 = -\sin\ \alpha \\
&\quad\quad a2 = s * \text{sinc}\ (\alpha/2) * \cos\ (\alpha/2) \\
&\quad\quad b0 = \sin\ \alpha \\
&\quad\quad b1 = \cos\ \alpha \\
&\quad\quad b2 = s * \text{sinc}\ (\alpha/2) * \sin\ (\alpha/2) \\
&= \texttt{let} \\
&\quad (x_{R_i}, y_{R_i}) = R_i \\
&\quad xv = \cos\ \alpha * x_{R_i} + (-\sin\ \alpha) * y_{R_i} + (s * \text{sinc}\ (\alpha/2) * \cos\ (\alpha/2)) \\
&\quad yv = \sin\ \alpha * x_{R_i} + \cos\ \alpha * y_{R_i} + (s * \text{sinc}\ (\alpha/2) * \sin\ (\alpha/2)) \\
&\quad \texttt{in}\ (xv, yv) && (69)
\end{aligned}$$

Similarly, $P^2_{i,s,\alpha}$ can be rewritten as well by applying the definitions and mathematical matrix operations.

$$P^2_{i,s,\alpha} = T(s,\alpha)R_i \tag{70}$$

$$= \begin{pmatrix} \cos\alpha & -\sin\alpha & s \cdot \text{sinc}\,\frac{\alpha}{2} \cdot \cos\frac{\alpha}{2} \\ \sin\alpha & \cos\alpha & s \cdot \text{sinc}\,\frac{\alpha}{2} \cdot \sin\frac{\alpha}{2} \end{pmatrix} \begin{pmatrix} x_{R_i} \\ y_{R_i} \\ 1 \end{pmatrix} \tag{71}$$

$$= \begin{pmatrix} \cos\alpha \cdot x_{R_i} + (-\sin\alpha) \cdot y_{R_i} + s \cdot \text{sinc}\,\frac{\alpha}{2} \cdot \cos\frac{\alpha}{2} \\ \sin\alpha \cdot x_{R_i} + \cos\alpha \cdot y_{R_i} + s \cdot \text{sinc}\,\frac{\alpha}{2} \cdot \sin\frac{\alpha}{2} \end{pmatrix} \tag{72}$$

By comparing (69) and (72), it is obvious that both expressions are isomorphic up to structure.

$$\mathsf{p2}\ R_i\ (s,\alpha) \cong P^2_{i,s,\alpha} \tag{73}$$

□

Next, it needs to be shown that $V^j_{i,s,\alpha}$ is isomorphic to $\mathsf{v}\ j\ R_i\ (s,\alpha)$. To show that, it comes in handy to proof first that $U^2_{i,s,\alpha}$ and $U^3_{i,s,\alpha}$ are isomorphic to $\mathsf{u2}\ R_i\ (s,\alpha)$ and $\mathsf{u3}\ R_i\ (s,\alpha)$, respectively. The method used for showing all three lemmas is similar to the previous proof with the difference that the proof for the correctness of v depends on the correctness of u2 and u3. Thus, the proofs are omitted here and (74) follows[5].

$$\mathsf{v} \cong V^j_{i,s,\alpha} \tag{74}$$

□

The proof continues with the rewrite of the stepTwo function (figure 18). First, the local definitions `interval` and p1p2v are rewritten. The arguments for the rewrite of the list comprehension is based on the *Haskell Report* [11, p. 22].

$$interval = \mathsf{mkInterval}\ (smin, smax)\ (amin, amax) \tag{75}$$

$$= [(s,a)|s \leftarrow [smin, smax], a \leftarrow [amin, amax]] \tag{76}$$

$$= \text{concatMap}\ (\lambda s \rightarrow \text{concatMap}\ (\lambda a \rightarrow [(s,a)])\ [amin, amax])\ [smin, smax] \tag{77}$$

$$= \text{concat}\ (\text{map}\ (\lambda s \rightarrow \text{concat}\ (\text{map}\ (\lambda a \rightarrow [(s,a)])\ [amin, amax]))\ [smin, smax]) \tag{78}$$

$$= \text{concat}\ (\text{map}\ (\lambda s \rightarrow [(s, amin), (s, amax)])\ [smin, smax]) \tag{79}$$

$$= [(smin, amin), (smin, amax), (smax, amin), (smax, amax)] \tag{80}$$

[5] The proofs can be found in Appendix C.

The step from (77) to (78) was justified with (81), which was proven via induction over xs in *Isabelle*.

$$\text{concatMap } f\ xs = \text{concat } (\text{map } f\ xs) \tag{81}$$

As one can see, `interval` is isomorphic to the set $\{s_{min}, s_{max}\} \times \{\alpha_{min}, \alpha_{max}\}$ assuming $(smin, smax) = (s_{min}, s_{max})$ and $(amin, amax) = (\alpha_{min}, \alpha_{max})$, which was proven in step one.

Further, `p1p2v` can be rewritten as (84).

$$p1p2v = p1\ :\ p2\ :\ \text{map } v\ [0..l-1] \tag{82}$$
$$= p1\ :\ p2\ ::\ v\ 0\ :\ \ldots\ :\ v\ (l-1)\ :\ [] \tag{83}$$
$$= [p1,\ p2,\ v\ 0\ ,\ \ldots\ ,\ v\ (l-1)] \tag{84}$$

Next, `stepTwo` can be unfolded. The **where**-clause definitions will be omitted, since they were just unfolded.

$$\begin{aligned}
&\text{stepTwo } (smin, smax)\ (amin, amax) \\
&= [f\ ri\ sa | sa \leftarrow interval, ri \leftarrow robotPoints, f \leftarrow p1p2v] && (85)\\
&= \texttt{let } ok\ sa\ = \\
&\qquad \texttt{let } ok'\ ri\ = \\
&\qquad\quad \texttt{let } ok''\ f\ =\ [f\ ri\ sa] \\
&\qquad\quad \texttt{in concatMap } ok''\ p1p2v \\
&\qquad \texttt{in concatMap } ok'\ robotPoints \\
&\quad \texttt{in concatMap } ok\ interval && (86)\\
&= \text{concatMap } (\lambda sa \rightarrow \text{concatMap } (\lambda ri \rightarrow \text{concatMap } (\lambda f \rightarrow [f\ ri\ sa])\ p1p2v)\ robotPoints)\ interval && (87)\\
&= \text{concat } (\text{map } (\lambda sa \rightarrow \text{concat } (\text{map } (\lambda ri \rightarrow \text{concat } (\text{map } (\lambda f \rightarrow [f\ ri\ sa])\ p1p2v))\ robotPoints))\ interval) && (88)\\
&= \text{concat } (\text{map } (\lambda sa \rightarrow \text{concat } (\text{map } (\lambda ri \rightarrow \text{concat } (\text{map } (\lambda f \rightarrow [f\ ri\ sa])\ p1p2v))\ robotPoints))\ interval) && (89)\\
&= \text{concat } (\text{map } (\lambda sa \rightarrow \text{concat } (\text{map } (\lambda ri \rightarrow \text{concat } ([[p1\ ri\ sa],\ [p2\ ri\ sa], \\
&\qquad [v\ 0\ ri\ sa],\ \ldots\ ,\ [v\ (l-1)\ ri\ sa]]))\ robotPoints))\ interval) && (90)\\
&= \text{concat } (\text{map } (\lambda sa \rightarrow \text{concat } (\text{map } (\lambda ri \rightarrow [p1\ ri\ sa,\ p2\ ri\ sa, \\
&\qquad v\ 0\ ri\ sa,\ \ldots\ ,\ v\ (l-1)\ ri\ sa])\ robotPoints))\ interval) && (91)\\
&= \text{concat } (\text{map } (\lambda sa \rightarrow \text{concat } (\text{map } (\lambda ri \rightarrow [p1\ ri\ sa,\ p2\ ri\ sa, \\
&\qquad v\ 0\ ri\ sa,\ \ldots\ ,\ v\ (l-1)\ ri\ sa])\ [r_1, \ldots, r_n]))\ interval) && (92)\\
&= \text{concat } (\text{map } (\lambda sa \rightarrow \text{concat } [xs_{r_1,sa}, \ldots, xs_{r_n,sa}])\ interval) && (93)\\
&= \text{concat } (\text{map } (\lambda sa \rightarrow xs_{r_1,sa} +\!\!+ \ldots +\!\!+ xs_{r_n,sa})\ interval) && (94)\\
&= \text{concat } (\text{map } (\lambda sa \rightarrow xs_{r_1,sa} +\!\!+ \ldots +\!\!+ xs_{r_n,sa})\ [(smin, amin), (smin, amax), (smax, amin), (smax, amax)]) && (95)\\
&= \text{concat } [xss_{(smin,amin)}, xss_{(smin,amax)}, xss_{(smax,amin)}, xss_{(smax,amax)}] && (96)\\
&= xss_{(smin,amin)} +\!\!+ xss_{(smin,amax)} +\!\!+ xss_{(smax,amin)} +\!\!+ xss_{(smax,amax)} && (97)
\end{aligned}$$

with

$$xs_{r_i,sa} = [p1\ r_i\ sa,\ p2\ r_i\ sa,\ v\ 0\ r_i\ sa,\ \ldots\ ,\ v\ (l-1)\ r_i\ sa] \tag{98}$$
$$xss_{(s,a)} = xs_{r_1,(s,a)} +\!\!+ \ldots +\!\!+ xs_{r_n,(s,a)} \tag{99}$$

Similarly, the final result of the specification of step two can be rewritten.

$$[[[P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]^{L-1}_{j=0}]^n_{i=0}]^{s_{max}}_{s_{min}}]^{\alpha_{max}}_{\alpha_{min}} \tag{100}$$

$$= \bigcup_{\substack{(s,\alpha)\\ \in I}} \{P^1_{1,s,\alpha}, \ldots, P^1_{n,s,\alpha}\} \cup [[[P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]^{L-1}_{j=0}]^n_{i=0}]^{s_{max}}_{s_{min}}]^{\alpha_{max}}_{\alpha_{min}} \tag{101}$$

$$= \bigcup_{\substack{(s,\alpha)\\ \in I}} \{P^1_{1,s,\alpha}, \ldots, P^1_{n,s,\alpha}\} \cup \bigcup_{\substack{(s,\alpha)\\ \in I}} \{P^2_{1,s,\alpha}, \ldots, P^2_{n,s,\alpha}\}$$

$$\cup [[[[V^j_{i,s,\alpha}]^{L-1}_{j=0}]^n_{i=0}]^{s_{max}}_{s_{min}}]^{\alpha_{max}}_{\alpha_{min}} \tag{102}$$

$$= \bigcup_{\substack{(s,\alpha)\\ \in I}} \bigcup_{i=1}^{n} P^1_{i,s,\alpha} \cup P^2_{i,s,\alpha} \cup \bigcup_{j=0}^{L-1} V^j_{1,s,\alpha} \tag{103}$$

with $I = \{s_{min}, s_{max}\} \times \{\alpha_{min}, \alpha_{max}\}$. By considering (65), (73) and (74), one can infer another relation of $xs_{r_i,(s,\alpha)}$:

$$xs_{r_i,(s,\alpha)} \cong xs'_{R_i,(s,\alpha)} \tag{104}$$

with $xs'_{R_i,(s,\alpha)} = \left\{P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, V^0_{i,s,\alpha}, \ \ldots \ , V^{L-1}_{i,s,\alpha}\right\}$. This implies another isomorphic relation:

$$xss_{(s,\alpha)} \cong xss'_{(s,\alpha)} \tag{105}$$

with $xss'_{(s,\alpha)} = \left\{xs'_{R_1,(s,\alpha)} \cup \ldots \cup xs'_{R_n,(s,\alpha)}\right\}$. Which in turn gives rise to (106) by considering the transformation of stepTwo in (97).

$$\text{stepTwo } (smin, smax)\ (amin, amax) \cong XS_{smin,smax,amin,amax} \tag{106}$$

with

$$XS_{s,s',a,a'} = xss'_{(s,a)} \cup xss'_{(s,a')} \cup xss'_{(s',a)} \cup xss'_{(s',a')} \tag{107}$$

Since $(\mathsf{smin}, \mathsf{smax}, \mathsf{amin}, \mathsf{amax}) \cong (s_{min}, s_{max}, \alpha_{min}, \alpha_{max})$ was shown in the verification of the first step, all four variables can be substituted in $XS_{smin,smax,amin,amax}$. After that, one can transform

$XS_{s_{min},s_{max},\alpha_{min},\alpha_{max}}$ by unfolding the definitions of $xss'_{s,\alpha}$ and $xs'_{R_i,s,\alpha}$.

$$
\begin{aligned}
& XS_{s_{min},s_{max},\alpha_{min},\alpha_{max}} \\
= \; & xss'_{(s_{min},\alpha_{min})} \cup xss'_{(s_{min},\alpha_{max})} \cup xss'_{(s_{max},\alpha_{min})} \cup xss'_{(s_{max},\alpha_{max})} && (108) \\
= \; & \left\{xs'_{R_1,(s_{min},\alpha_{min})} \cup \ldots \cup xs'_{R_n,(s_{min},\alpha_{min})}\right\} \\
& \cup \left\{xs'_{R_1,(s_{min},\alpha_{max})} \cup \ldots \cup xs'_{R_n,(s_{min},\alpha_{max})}\right\} \\
& \cup \left\{xs'_{R_1,(s_{max},\alpha_{min})} \cup \ldots \cup xs'_{R_n,(s_{max},\alpha_{min})}\right\} \\
& \cup \left\{xs'_{R_1,(s_{max},\alpha_{max})} \cup \ldots \cup xs'_{R_n,(s_{max},\alpha_{max})}\right\} && (109) \\
= \; & \Big\{\left\{P^1_{1,s_{min},\alpha_{min}}, P^2_{1,s_{min},\alpha_{min}}, V^0_{1,s_{min},\alpha_{min}}, \ \ldots \ , V^{L-1}_{1,s_{min},\alpha_{min}}\right\} \\
& \qquad \cup \ldots \cup \left\{P^1_{n,s_{min},\alpha_{min}}, P^2_{n,s_{min},\alpha_{min}}, V^0_{n,s_{min},\alpha_{min}}, \ \ldots \ , V^{L-1}_{n,s_{min},\alpha_{min}}\right\}\Big\} \\
& \cup \Big\{\left\{P^1_{1,s_{min},\alpha_{max}}, P^2_{1,s_{min},\alpha_{max}}, V^0_{1,s_{min},\alpha_{max}}, \ \ldots \ , V^{L-1}_{1,s_{min},\alpha_{max}}\right\} \\
& \qquad \cup \ldots \cup \left\{P^1_{n,s_{min},\alpha_{max}}, P^2_{n,s_{min},\alpha_{max}}, V^0_{n,s_{min},\alpha_{max}}, \ \ldots \ , V^{L-1}_{n,s_{min},\alpha_{max}}\right\}\Big\} \\
& \cup \Big\{\left\{P^1_{1,s_{max},\alpha_{min}}, P^2_{1,s_{max},\alpha_{min}}, V^0_{1,s_{max},\alpha_{min}}, \ \ldots \ , V^{L-1}_{1,s_{max},\alpha_{min}}\right\} \\
& \qquad \cup \ldots \cup \left\{P^1_{n,s_{max},\alpha_{min}}, P^2_{n,s_{max},\alpha_{min}}, V^0_{n,s_{max},\alpha_{min}}, \ \ldots \ , V^{L-1}_{n,s_{max},\alpha_{min}}\right\}\Big\} \\
& \cup \Big\{\left\{P^1_{1,s_{max},\alpha_{max}}, P^2_{1,s_{max},\alpha_{max}}, V^0_{1,s_{max},\alpha_{max}}, \ \ldots \ , V^{L-1}_{1,s_{max},\alpha_{max}}\right\} \\
& \qquad \cup \ldots \cup \left\{P^1_{n,s_{max},\alpha_{max}}, P^2_{n,s_{max},\alpha_{max}}, V^0_{n,s_{max},\alpha_{max}}, \ \ldots \ , V^{L-1}_{n,s_{max},\alpha_{max}}\right\}\Big\} && (110)
\end{aligned}
$$

If the inner structure of the multiset in (110) is dismissed and it is just taken as a multiset of tuples, additional rewrites can be performed by rearranging the elements and applying definitions from set theory.

$$
\begin{aligned}
& XS_{s_{min},s_{max},\alpha_{min},\alpha_{max}} \\
\cong \; & \left\{P^1_{1,s_{min},\alpha_{min}}, P^2_{1,s_{min},\alpha_{min}}, V^0_{1,s_{min},\alpha_{min}}, \ \ldots \ , V^{L-1}_{1,s_{min},\alpha_{min}}\right\} \\
& \qquad \cup \ldots \cup \left\{P^1_{n,s_{min},\alpha_{min}}, P^2_{n,s_{min},\alpha_{min}}, V^0_{n,s_{min},\alpha_{min}}, \ \ldots \ , V^{L-1}_{n,s_{min},\alpha_{min}}\right\} \\
& \cup \left\{P^1_{1,s_{min},\alpha_{max}}, P^2_{1,s_{min},\alpha_{max}}, V^0_{1,s_{min},\alpha_{max}}, \ \ldots \ , V^{L-1}_{1,s_{min},\alpha_{max}}\right\} \\
& \qquad \cup \ldots \cup \left\{P^1_{n,s_{min},\alpha_{max}}, P^2_{n,s_{min},\alpha_{max}}, V^0_{n,s_{min},\alpha_{max}}, \ \ldots \ , V^{L-1}_{n,s_{min},\alpha_{max}}\right\} \\
& \cup \left\{P^1_{1,s_{max},\alpha_{min}}, P^2_{1,s_{max},\alpha_{min}}, V^0_{1,s_{max},\alpha_{min}}, \ \ldots \ , V^{L-1}_{1,s_{max},\alpha_{min}}\right\} \\
& \qquad \cup \ldots \cup \left\{P^1_{n,s_{max},\alpha_{min}}, P^2_{n,s_{max},\alpha_{min}}, V^0_{n,s_{max},\alpha_{min}}, \ \ldots \ , V^{L-1}_{n,s_{max},\alpha_{min}}\right\} \\
& \cup \left\{P^1_{1,s_{max},\alpha_{max}}, P^2_{1,s_{max},\alpha_{max}}, V^0_{1,s_{max},\alpha_{max}}, \ \ldots \ , V^{L-1}_{1,s_{max},\alpha_{max}}\right\} \\
& \qquad \cup \ldots \cup \left\{P^1_{n,s_{max},\alpha_{max}}, P^2_{n,s_{max},\alpha_{max}}, V^0_{n,s_{max},\alpha_{max}}, \ \ldots \ , V^{L-1}_{n,s_{max},\alpha_{max}}\right\} && (111) \\
= \; & \bigcup_{\substack{(s,\alpha)\\ \in I}} \bigcup_{i=1}^{n} P^1_{i,s,\alpha} \cup P^2_{i,s,\alpha} \cup \bigcup_{j=0}^{L-1} V^j_{1,s,\alpha} && (112)
\end{aligned}
$$

(112) is equivalent to the transformation of the specification of the second step in (103). Because (112) is isomorphic to $XS_{s_{min},s_{max},\alpha_{min},\alpha_{max}}$ and $XS_{s_{min},s_{max},\alpha_{min},\alpha_{max}}$ is isomorphic to $\mathsf{stepTwo}(s_{min}, s_{max})\ (\alpha_{min}, \alpha_{max})$, one can finally conclude that the implementation of the second

step of the algorithm calculates a correct sequence of points.

$$\mathsf{stepTwo}(s_{min}, s_{max})\ (\alpha_{min}, \alpha_{max}) \cong [[[P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]_{j=0}^{L-1}]_{i=0}^{n}]_{s_{min}}^{s_{max}}]_{\alpha_{min}}^{\alpha_{max}} \tag{113}$$

□

It remains to be shown that the implementation of the buffer radius calculation `calcBuffer` is correct as well. The definition of the relevant specification and implementation is shown in figure 19.

$$q = q^A + q^B$$

$$q^A = \frac{1}{6}\Big(\frac{\alpha_{max} - \alpha_{min}}{2}\Big)^2 \max\{|s_{min}|; |s_{max}|\}$$

$$q^B = \Big(1 - \cos\frac{\alpha_{max} - \alpha_{min}}{2}\Big) \max_i |R_i|$$

```
calcBuffer :: (Float, Float) → (Float, Float) → Float
calcBuffer (sMin, sMax) (aMin, aMax) =
  1/6 * (av*av) * maxS + (1−cos av) * maxR
  where
    av = (aMax−aMin)/2
    maxS = max (abs sMin) (abs sMax)
    maxR = d

d :: Float
d = maximum (map vLen robotPoints)
  where
    vLen (x,y) = sqrt (x*x + y*y)
```

Figure 19: Definition of `calcBuffer`.

The relation $\mathsf{d} \cong D$ was shown earlier on page 26. Because D is defined as $\max_i |R_i|$, d is isomorphic to the local definition maxR as well. Hence, (114) follows, since both definition q and `calcBuffer` are almost identical if every locally bound name within `calcBuffer` is substituted with their local definition.

$$\mathsf{calcBuffer}(s_{min}, s_{max})\ (\alpha_{min}, \alpha_{max}) \cong q \tag{114}$$

□

The composition of the calculated list of points and the buffer radius are the result of the main function `safetyZone` as displayed in figure 13. Since (113) and (114) were shown previously, the composition is certainly isomorphic to the composition of $[[[P^1_{i,s,\alpha}, P^2_{i,s,\alpha}, [V^j_{i,s,\alpha}]_{j=0}^{L-1}]_{i=0}^{n}]_{s_{min}}^{s_{max}}]_{\alpha_{min}}^{\alpha_{max}}$ and q, which is defined as the end result of the specification (see (12)). It is thereby concluded that the implementation of the algorithm calculates a result as defined by the specification under the made assumptions.

□

The whole pen and paper proof including its formalization was done over a period of five weeks by the author of this thesis.

3.2.3 Additional Assurance through Types

As seen in the previous section, most of the verification process was based on *equational reasoning*, where a (sub-)expression is substituted with an equivalent expression, e.g. `map f ∘ map g` can be replaced with `map (f ∘ g)`. The exception to that is the proof for the binary search in the `findIndex` function, which is proven by induction over the difference of the range indices. This thesis argues that the former proof method is considerably less likely to contain human-induced logical errors, since the method relies completely on the almost mechanically substitution of definitions, which can be easily checked for errors. Whereas the later could contain undiscovered mathematical fallacies. Hence, it was decided to support the proof of correctness of `findIndex` by an additional machine-aided proof.

Naturally, there are many different options and approaches for a machine-supported proof. The first intuitive approach would be to use a generic theorem prover like *Isabelle*, which was used in the previous section to show trivial lemmas. Nevertheless, this was not the preferred solution, because of the previously mentioned inexperience of the author with *Isabelle*. Furthermore, it would be necessary to translate all involved functions from the implementation language to *HOL*. This would either open up another possible source of errors if the translation itself is not verified, or mean an additional proving-effort by verifying the translation.
Therefore, a better option would operate on the actual *Haskell* implementation of `findIndex`. One way of doing this would be to embed the specification directly into the implementation by utilizing *dependent types*. Dependent types are types that contain value expressions, i.e. types that are dependent on values [2, p. 1-3]. Unfortunately, *Haskell*'s type system does not support dependent types. Nonetheless, there are ways to simulate dependent types by combining several language extensions that extend or modify the type system [6, 12]. Additionally, there are efforts to integrate them fully into *Haskell* [5]. However, all these approaches require a refactoring of the existing implementation. In addition to that, in their current state they arguably make the program less readable. Hence, it was decided to not verify `findIndex` using dependent types, but rather *refinement types* as they are provided by *Liquid Haskell*.

Refinement types are types that include a predicate, which needs to hold for every instance of this type [24]. *Liquid Haskell* is a standalone application that implements these types by checking if the implementation and the input of functions are consistent with their refinements types. The provided types extend *Haskell*'s type system but they are not interfering with it, i.e. they are not known by the compiler. Rather, the refinement types are written in specially formatted *Haskell* comments, which can be interpreted by *Liquid Haskell*. Hence, a compilation would still be possible, even if *Liquid Haskell* would reject the program [23, p. 1-2]. In addition to the type check, termination of functions is checked as well. The decreasing measure for this check can be specified by the programmer [23, p. 5-6].

As stated previously, *Liquid Haskell* was used to verify the binary search algorithm used by `findIndex`. The verification was done in isolation, because otherwise all other parts of the program, which depend on `findIndex`, would have been needed to be verified by *Liquid Haskell* as well. To further simplify the matter, the braking measurements were reduced to a sequence of just the velocities. The necessary adaptions that this change caused were implemented as well.

Figure 20 shows the modified version of `findIndex` with added refinement types. The first necessary refinement concerns the variable `brakingMeasurements`[6], whose type is not only `Vector (Double, Double)` anymore, but a `Vector (Double, Double)` with a fixed length, three in this case[7]. Next, the *measure function* `at` is specified. Measurements are uninterpreted functions, that can generate refinement types for data constructors [23, p. 3]. In this case, the only thing *Liquid Haskell* can state about `at` is its referential transparency, since no implementation of the measure was supplied. Still, `at` is necessary to create a connection between the vector accessor function `(!)` on the expression level and the refinements on the type level. This connection is established in the next definition, which creates an assumption that states that `x ! i` results in value `v`, which is within the refinements equal to `at x i`. This is only the case if `i` lies within the ranges of the vector, i.e. `x ! i` $\neq \bot$.

Following that, `findIndex` and its local definition `go` get annotated with additional refinement types. The refinement type of `findIndex` captures the specification: if given a v, which lies within the ranges of the braking measurements, the function results in an index ind such that $v_{ind} \leq v \leq v_{ind+1}$. `go` captures the specification as well but adds predicates in the refinements of the argument types about the relationship between the arguments. In order to let *Liquid Haskell* show termination of `go` the decreasing measure $imax - imin$ had to be supplied.

All these additional types were developed incrementally, starting with just the refinement type of the result of `findIndex`. From there, every other refinement type was added or modified because of the type mismatch error messages of *Liquid Haskell*. The resulting types as seen in figure 20 are verified and accepted by *Liquid Haskell*. It is therefore assumed that the provided implementation of `findIndex` is correct under the assumptions made. Furthermore, this reinforces the confidence in the correctness of the verification process.

The additional verification of `findIndex` took about two weeks to complete. This includes the time to learn the basics of the syntax and semantics of *Liquid Haskell*.

[6] For brevity renamed to `bms` here.
[7] In the isolation module the type is just `Vector Double`.

```
module Liquid where

import Data.Vector (Vector, (!), fromList, length)
import Prelude hiding (length)

{-@ bms :: {vec : Vector Double | vlen vec == 3 } @-}
bms :: Vector Double
bms = fromList [0, 203.86, 535.131]

{-@ measure at :: Vector a → Int → a @-}
{-@ assume (!) :: x:Vector a → i:{Nat | 0 ≤ i && i < vlen x}
                  → {v:a | v = at x i} @-}

{-@ findIndex ::
      v:{d:Double | (at bms 0) ≤ d && d ≤ (at bms (vlen bms-1)) }
      → {ind:Int | (at bms ind) ≤ v && v ≤ (at bms (ind + 1)) }
 @-}
findIndex :: Double → Int
findIndex v = go 0 (length bms - 1) where
  {-@ go ::
        imin:{im:Int | im < vlen bms && (at bms im) ≤ v}
        → imax:{imm:Int | imm > imin && imm < vlen bms
                                     && v ≤ (at bms imm)}
        → {indRes:Int | (at bms indRes) ≤ v && v ≤ (at bms (indRes+1))}
        / [imax-imin]
   @-}
  go imin imax = let i = (imax+imin) `div` 2 in
    if imax-imin > 1
    then
      if bms ! i < v
      then go i imax
      else go imin i
    else imin
```

Figure 20: Implementation of `findIndex` with refinements types. Accepted by *Liquid Haskell*.

3.3 Performance Comparison

The presented algorithm is meant to be run on an autonomous vehicle in real time, such that dangerous collisions with other objects can be avoided. This shows that performance plays an equally important role as correctness, since correctness alone does not matter much if the algorithm is not applicable.

In theory, the first step of the functional implementation `stepOne` has a time complexity of $\mathcal{O}(\log b)$ with b being the number of previously recorded braking measurements, because every computation in the first step is bound by a constant number with the exception of the binary search algorithm in `findIndex`. Since the effect of an increase in the number of braking measurements is rather small [20, p. 22-23] and taking accurate braking measurements is an effortful task, it is rather unlikely that this number grows significantly. Hence, it is assumed that the first part of the algorithm has a time complexity of $\mathcal{O}(1)$. The second part of the algorithm `stepTwo` has a time complexity of $\mathcal{O}(n \cdot m)$ with n being the number of contour points and m being the approximation points per circular arc L. The reason for this classification is the growth of the list comprehension in `stepTwo`. Its size and therefore the number of computations can be described by the function $f(n, m) = 4 \cdot n \cdot (2 + m)$. $f(n, m)$ cannot be bound by the term $c \cdot (n + m)$ for some constant c, but can be bound by the function $g(n, m) = c \cdot (n \cdot m)$ for all $m > m_0, n > n_0$ with $c = 12, m_0 = 0$ and $n_0 = 0$. All other operations as well as the `calcBuffer` function have a constant time complexity. Hence, the overall time complexity of the implementation is $\mathcal{O}(n \cdot m)$.

In real-time systems theoretical time complexity is often not as important as actual runtime. Therefore, the actual runtime of the *Haskell* implementation as well as the original *C* implementation were measured. All tests were done on a *2.7GHz dual-core Intel Core i5 processor*. Both programs were compiled with the highest available optimization options without breaking compliance, which means `-O2` for *GHC* and `-O3` for *Clang*. Since *Haskell* uses a lazy evaluation strategy by default [11, p. 75], it was necessary to ensure that the results during the performance tests are fully evaluated. To make sure that this is the case, the *criterion* library was used. The library provides an interface for measuring *Haskell* functions. Additionally, it provides a way of ensuring that the result of a pure function application is evaluated to its normal form [17]. In contrast to the original implementation where only the function `schutzfeld_berechnen` up to the finished call to the `schutzfeld_huelle_plus_radius` function was measured, the complete *Haskell* implementation in the form of `safetyZone` was measured[8].

Figure 21 and 22 show the results of the performance tests of the original and the functional implementation, respectively. While the aforementioned figures show the complete result, figure 23 depicts a more detailed comparison for two fixed numbers of approximation points per arc. Both

[8] The function names are referring to the publicly available *safety component* library accessible online at `https://www-cps.hb.dfki.de/sams/software.en.html` (Accessed: 03.07.2018).

implementations seem to grow in a similar manner, but the functional implementation is always slower than its imperative counterpart. In detail, the *Haskell* implementation is 7% ($|\mathfrak{R}| = 4, L = 8$) to 308% ($|\mathfrak{R}| = 25, L = 25$) slower than the C implementation. The decline of the time factor with increasing $|\mathfrak{R}|$ and L is expected, since the implementation is not optimized and performs calculations that are not necessary as mentioned on page 17. In absolute numbers this translates to a runtime of under 2 ms with a rather detailed vehicle shape $|\mathfrak{R}| = 25$ and a closed-meshed approximation of the arcs $L = 25$, which should still be sufficient at least for slow-moving vehicles. For rather realistic vehicle shapes ($|\mathfrak{R}| < 10$, as stated by the original paper [20, p. 23]) and a conservative approximation of arcs ($L = 8$) the algorithm runs in under 0.17 ms, which would correspond to a traveled distance of 4.72 mm for a vehicle traveling with 100 km/h.

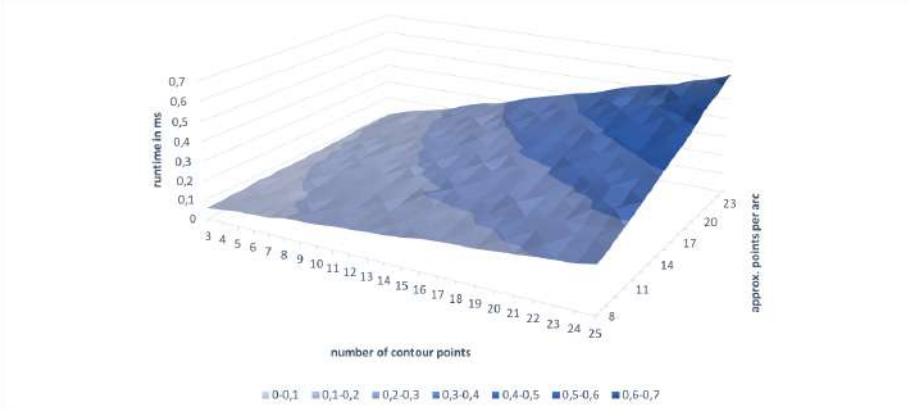


Figure 21: Runtime of the C implementation.

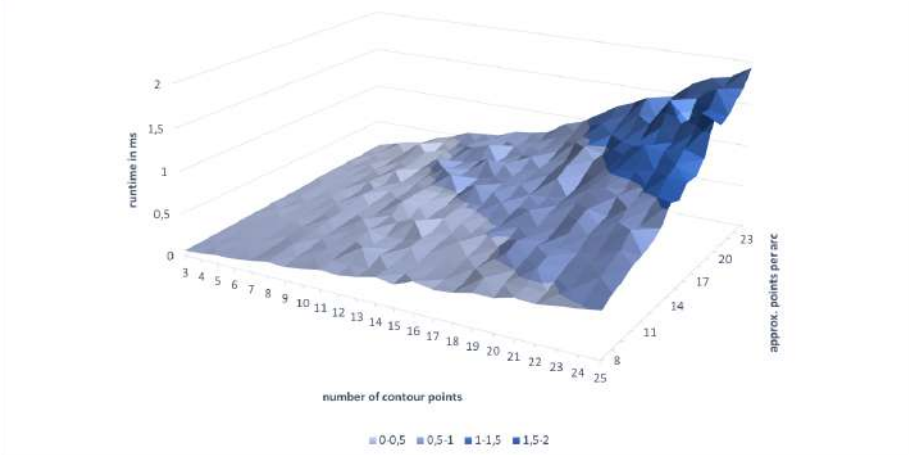
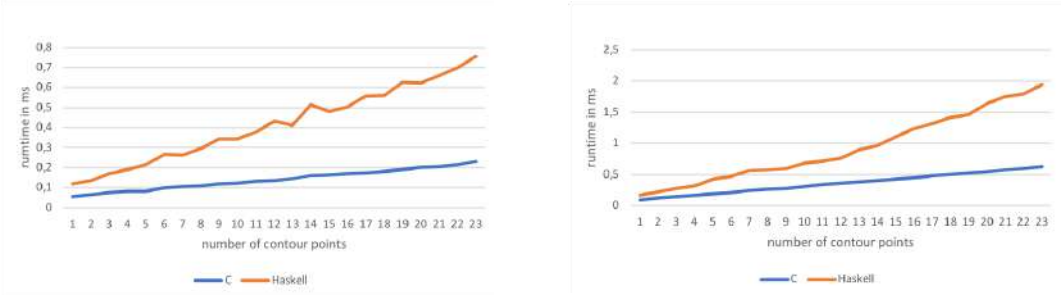


Figure 22: Runtime of the *Haskell* implementation.



(a) $L = 8$ (b) $L = 25$

Figure 23: Runtime comparison for two fixed numbers of approximation points per arc.

4 Conclusion, Discussion and Outlook

This thesis investigated whether an implementation in a functional, as opposed to an imperative programming language, would improve the speed of the verification process, assuming that its specification is mathematically formalized. To explore this hypothesis a case study was performed. The topic of the case study was the development of a safety zone algorithm for autonomous vehicles. The original algorithm was implemented in the imperative programming language *C* using the *MISRA-C* guidelines. Afterwards, the implementation was formally verified using mostly the interactive theorem prover *Isabelle* [20].
Based on the specification of the case study, an implementation in the pure functional programming language *Haskell* was developed. After that, the implementation was verified via a pen and paper proof. Furthermore, critical parts of the program were verified again with the help of *Liquid Haskell*, which is a standalone extension to *Haskell's* type system. An initial attempt to verify the implementation in *Isabelle* failed due to the inexperience of the author and time constraints of this work. Finally, the runtime performance of both implementations was measured.

A comparison of the time it took to verify both implementations shows that the functional implementation was indeed faster to verify. Prior to the verification of the *C* implementation the domain needed to be modeled in *Isabelle*. This development took 5 month and was done mostly by a mathematician [25, p.153]. After that, the actual implementation was verified over a period of 6 month by a member of the SAMS project [25, p.170], whereas the functional implementation was verified within 5 weeks on paper by the author of this thesis. Additional 2 weeks were necessary to proof the correctness of the implementation of the binary search algorithm with *Liquid Haskell*. However, the performance comparison shows that the *Haskell* implementation is roughly up to 3 times slower than the original version.

Yet, the correlation between the time spans of the verification processes and the programming paradigm might not imply causality. There are several points concerning the results that need to be critically discussed. First, *equational reasoning* can be a simple but powerful technique to prove correctness of pure functional programs as shown by previous chapters. It is also not possible to use *equational reasoning* outside of referential transparent functions. Since idiomatic *C* code makes heavy use of effectful procedures, which are by definition referential opaque, it is not possible to utilize *equational reasoning* within idiomatic *C* code. Hence, *equational reasoning* is a proof technique which is most likely not applicable when verifying imperative implementations.
On the other hand, the realized case study may just be a perfectly fitting example for an ideal verification of a functional implementation, because the specification is just based on terms, which could be directly implemented. Nevertheless, other algorithms may have specifications, which make a direct and efficient implementation impossible. An example of such an algorithm could be one that relies heavily on stateful computation, e.g. simulation of a Turing machine. Further research is necessary to inspect if functional implementations lead to an easier verification process in such

cases as well.

Another point that needs to be addressed is the comparability of both verification processes. While the verification of the original implementation was done with the help of an interactive theorem prover, the verification in this thesis was only partly machine-aided and was mostly done on pen and paper. This may lead to comparability issues, since it is not clear in which time the functional implementation could have been proven in *Isabelle* or in what time span the imperative implementation could have been verified with pen and paper. Another problem is the comparability of the backgrounds of the authors, since both kinds of verification processes (machine-assisted or not) rely heavily on the ideas and methodology of the user. Thus, the background knowledge and experience of the user introduces another independent variable that should be considered in future research when a general statement regarding the time of the verification process is the goal.

Furthermore, the additional machine-aided proof of correctness of the binary search implementation encoded in *Liquid Haskell's* refinement type system is not unique to *Haskell* or to functional programming languages in general. While the usage of refinement types to automatically prove properties in a lazy language like *Haskell* may be exclusive to *Liquid Haskell*, the general notion of annotating programs with properties that must hold is not. For instance, there is the imperative programming language *Dafny* developed by *Microsoft Research*. The language features a program verifier that automatically checks if program properties (e.g. measurements or loop invariants) that the user can specify, hold [10]. Further, there are also formal verification environments even to existing imperative languages such as the *Frama-C* platform [9] or the *Verifying C Compiler (VCC)* [19]. Both frameworks allow a direct encoding of correctness properties into existing *C* code with the ability to automatically verify said properties. Additionally, some of the authors of *Liquid Haskell* worked previously on *CSolve*, a framework which implements liquid types for *C* [18]. Therefore, even though *Liquid Haskell* was beneficial to the verification process in this case, the general usage of automatic program verifiers is not bound to the programming paradigm.

With regards to the runtime, the performance tests show that the developed functional implementation, even in its current state, should be fast enough to compute safety zones in real time if one considers the fact that the autonomous vehicle for which the original algorithm was developed, only needs to perform a safety zone calculation every 40 ms [25, p.150]. However, the performance tests show also that the original implementation is up to 3 times faster. This definitely can be improved algorithmically, e.g. not computing points that are already present in the safety zone, and technically, e.g. using *GHC's* unboxed primitive data types instead of the boxed primitives [7]. Additionally, there may be computations within the current implementation that would benefit from a strict evaluation, but in order to detect these further analysis of the runtime and space behavior is necessary.

At last, there is the issue of the scope of the verification. All variables that are specific to a vehicle configuration are just assumed to be correct. This is practical and reasonable for a verification setting, but in an actual production environment this could still lead to incorrect results, which could ultimately endanger human lives. Therefore, a more safe solution to ensure that entities in the code meet the assumption would be welcome. Additional tools like *Liquid Haskell* at first seem like a solution to solve that problem, but since they are external extensions to the compiler, they do not need to be run to compile a program. Hence, there still remains the risk that developers intentionally or unintentionally do not run these external checkers. Thus, a solution that is integrated into the compiler would be favorable, e.g. *dependent types*.

Ultimately, the results of this thesis show that in this particular case under the chosen methods the functional implementation was indeed faster to verify than its imperative implementation with its verification methods. In contrast, the runtime performance of the reimplementation was significantly slower than its counterpart, while still being applicable in production environments. However, due to the previously mentioned reasons the assertion that functional implementations in general are faster to verify than imperative implementations cannot be derived. Future research, in the form of empirical studies, are necessary to investigate this claim.

Bibliography

[1] *Verification Techniques*, pages 127–166. Springer New York, New York, NY, 2004. ISBN 978-0-387-21551-8. doi: 10.1007/0-387-21551-4_4. URL https://doi.org/10.1007/0-387-21551-4_4.

[2] Ana Bove and Peter Dybjer. Language engineering and rigorous software development. chapter Dependent Types at Work, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03152-6. doi: 10.1007/978-3-642-03153-3_2. URL http://dx.doi.org/10.1007/978-3-642-03153-3_2.

[3] DFKI Bremen. SAMS project website. https://www-cps.hb.dfki.de/sams/index.en.html, . Accessed: 13.04.2018.

[4] DFKI Bremen. SAMS project website - TÜV reports. https://www-cps.hb.dfki.de/sams/reports.en.html, . Accessed: 13.04.2018.

[5] Richard A. Eisenberg. Dependent types in haskell: Theory and practice. *CoRR*, abs/1610.07978, 2016. URL http://arxiv.org/abs/1610.07978.

[6] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364522. URL http://doi.acm.org/10.1145/2364506.2364522.

[7] GHC Team. Glasgow haskell compiler user's guide - unboxed types and primitive operations. http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#unboxed-types-and-primitive-operations, 2018. Accessed: 12.06.2018.

[8] GHC Team. Glasgow haskell compiler user's guide - overloaded lists. http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#overloaded-lists, 2018. Accessed: 03.05.2018.

[9] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015. ISSN 1433-299X. doi: 10.1007/s00165-014-0326-7. URL https://doi.org/10.1007/s00165-014-0326-7.

[10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

[11] Simon Marlow. Haskell 2010 language report. https://www.haskell.org/definition/haskell2010.pdf, 2010. Accessed: 03.05.2018.

[12] Conor McBride. Faking it simulating dependent types in haskell. *J. Funct. Program.*, 12(5):375–392, July 2002. ISSN 0956-7968. doi: 10.1017/S0956796802004355. URL http://dx.doi.org/10.1017/S0956796802004355.

[13] MIRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems, October 2004. URL www.misra.org.uk.

[14] John C. Mitchell and Krzysztof Apt. *Concepts in Programming Languages*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521780985.

[15] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[16] Oracle Corporation. What's new in jdk 8. https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html, 2014. Accessed: 30.07.2018.

[17] Bryan O'Sullivan. A criterion tutorial. http://www.serpentine.com/criterion/tutorial.html, 2014. Accessed: 03.07.2018.

[18] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 131–144, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706316. URL http://doi.acm.org/10.1145/1706299.1706316.

[19] Wolfram Schulte. Vcc: Contract-based modular verification of concurrent c. In *31st International Conference on Software Engineering, ICSE 2009*. IEEE Computer Society, January 2008. URL https://www.microsoft.com/en-us/research/publication/vcc-contract-based-modular-verification-of-concurrent-c/.

[20] Holger Täubig, Udo Frese, Christoph Hertzberg, Christoph Lüth, Stefan Mohr, Elena Vorobev, and Dennis Walter. Guaranteeing functional safety: design for provability and computer-aided verification. *Autonomous Robots*, 32(3):303–331, Apr 2012. ISSN 1573-7527. doi: 10.1007/s10514-011-9271-y. URL https://doi.org/10.1007/s10514-011-9271-y.

[21] TIOBE Software BV. Tiobe index. https://www.tiobe.com/tiobe-index/, 2018. Accessed: 30.07.2018.

[22] University of Cambridge and Technische Universität München. Isabelle website. http://isabelle.in.tum.de/documentation.html. Accessed: 16.04.2018.

[23] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: Experience with refinement types in the real world. *SIGPLAN Not.*, 49(12):39–51, September 2014. ISSN 0362-1340. doi: 10.1145/2775050.2633366. URL http://doi.acm.org/10.1145/2775050.2633366.

[24] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282, August 2014. ISSN 0362-1340. doi: 10.1145/2692915.2628161. URL http://doi.acm.org/10.1145/2692915.2628161.

[25] Dennis Walter. *A formal verification environment for use in the certification of safety-related C programs*. PhD thesis, Bremen, Univ., Diss., 2010, Bremen, 2010. URL http://nbn-resolving.de/urn:nbn:de:gbv:46-00101756-18. Online-Ressource (229 S., 2,27 MB).

[26] Waymo LLC. https://storage.googleapis.com/sdc-prod/v1/safety-report/Safety%20Report%202018.pdf, 2018.

Appendix

A Complete Haskell Implementation

Main.hs

```
module Main where

import          Control.Monad
import          ConvexHull
import qualified SafetyZone                as S
import          System.Environment

main :: IO ()
main = do
  args <- getArgs
  let
    [vmin,vmax,wmin,wmax] = map read args
    (zone, q) = S.safetyZone (vmin,vmax) (wmin,wmax)
  print "Points:"
  mapM_ print zone
  putStrLn $ "Q: " ++ show q
```

SafetyZone.hs

```
{-# LANGUAGE OverloadedLists #-}
module SafetyZone where

import          Data.Vector (Vector, (!))

-- Configurations
type Velocity = Float
type Distance = Float
type Time = Float
type SafetyZone = ([Point], BufferRadius)
type BufferRadius = Float
type Point = (Float, Float)

brakingMeasurements :: Vector (Velocity, Distance)
brakingMeasurements = [(0, 0), (203.86, 113.5), (535.131, 365.0)]
```

```
-- delta t
latency :: Time
latency = 0.06

robotPoints :: [Point]
robotPoints = [(-233.5,-162.5), (193.5,-162.5), (193.5,162.5), (-233.5,162.5)]

l :: Float
l = fromIntegral (8 :: Int)

-- D
d :: Float
d = maximum (map vLen robotPoints)
  where
    vLen (x,y) = sqrt (x*x + y*y)

infixl 6 <->
(<->) :: Num a => (a,a) -> (a,a) -> (a,a)
(x1,y1) <-> (x2,y2) = (x1-x2,y1-y2)

infixl 6 <+>
(<+>) :: Num a => (a,a) -> (a,a) -> (a,a)
(x1,y1) <+> (x2,y2) = (x1+x2,y1+y2)

infixl 7 <.>
(<.>) :: Num a => a -> (a,a) -> (a,a)
s <.> (x,y) = (s*x,s*y)

infixl 7 <..>
(<..>) :: ((Float, Float), (Float, Float)) -> Point -> Point
((a0,a1),(b0,b1)) <..> (x,y) = (a0*x + a1*y, b0*x + b1*y)

infix 7 <:.>
(<:.>) :: ((Float, Float, Float), (Float, Float, Float)) -> Point -> Point
((a0,a1,a2),(b0,b1,b2)) <:.> (x,y) = (a0*x+a1*y+a2, b0*x+b1*y+b2)



safetyZone :: (Float, Float) -> (Float, Float) -> SafetyZone
```

```
safetyZone vMinMax wMinMax =
  let (sMinMax,aMinMax) = stepOne vMinMax wMinMax
  in (stepTwo sMinMax aMinMax, calcBuffer sMinMax aMinMax)

stepOne :: (Float, Float) -> (Float, Float) -> ((Float,Float),(Float, Float))
stepOne (vMin, vMax) (wMin, wMax) = ((sMin,sMax), (aMin,aMax)) where
  candidates = mkCandidates (vMin, vMax) (wMin, wMax)
  (sCandidates, αCandidates) = unzip (map bm candidates)
  sMin = minimum sCandidates
  sMax = maximum sCandidates
  aMin = minimum αCandidates
  aMax = maximum αCandidates

bm :: (Float, Float) -> (Float, Float)
bm (v,w) = let ts' = ts (v,w) in ((ts'+latency)*v,(ts'+latency)*w)

mkCandidates :: (Float, Float) -> (Float, Float) -> [(Float,Float)]
mkCandidates (vMin, vMax) (wMin, wMax) = [(v,w) | v <- velos, w <- omega] where
  velos
    | 0 ≥ vMin && 0 ≤ vMax = [vMin,0,vMax]
    | otherwise = [vMin,vMax]
  omega
    | 0 ≥ wMin && 0 ≤ wMax = [wMin,0,wMax]
    | otherwise = [wMin,wMax]

ts :: (Float, Float) -> Float
ts (v,w)
  | vs ≥ vm = sm / (vm*vm*vm) * vs * vs
  | vs ≤ v1 = s1 / v1
  | otherwise =
    let
      j' = findIndex vs
      (vj',sj') = brakingMeasurements ! j'
      (vj,sj) = brakingMeasurements ! (j'+1)
    in
      (sj' + ((sj-sj')/(vj-vj')) * (vs-vj')) / vs
    where
      vs = sqrt (v*v+(d*d)*(w*w))
      (vm,sm) = brakingMeasurements ! (length brakingMeasurements - 1)
```

```
      (v1,s1) = brakingMeasurements ! 1

findIndex :: Velocity -> Int
findIndex v = go 0 (length brakingMeasurements-1) where
  go imin imax = let i = (imax+imin) `div` 2 in
    if imax-imin > 1
    then if fst (brakingMeasurements ! i) < v
         then go i imax
         else go imin i
    else imin

stepTwo :: (Float,Float) -> (Float, Float) -> [Point]
stepTwo (smin,smax) (amin,amax) = [ f ri sa | sa <- interval, ri <- robotPoints, f <-
     p1p2v ]
  where
    interval = mkInterval (smin,smax) (amin,amax)
    p1p2v = p1 : p2 : map v [0..l-1]

mkInterval :: (Float, Float) -> (Float, Float) -> [(Float, Float)]
mkInterval (smin,smax) (amin,amax) = [ (s,a) | s <- [smin,smax], a <- [amin,amax] ]

p1 :: Point -> (Float, Float) -> Point
p1 = const

p2 :: Point -> (Float, Float) -> Point
p2 ri (s,a) = tMatrix s a <.> ri

u2 :: Point -> (Float, Float) -> Point
u2 ri (s,a) = tMatrix (s/l) (a/l) <.> ri

u3 :: Point -> (Float, Float) -> Point
u3 u1@ri sa@(s,a) = u1 <+> qMatrix (a/l) <..> (0.5 <.> (u2 ri sa <-> u1))

v :: Float -> Point -> (Float, Float) -> Point
v j ri sa@(s,a) = tMatrix ((j*s)/l) ((j*a)/l) <.> u3 ri sa

tMatrix :: Float -> Float -> ((Float, Float, Float), (Float, Float, Float))
tMatrix s a = ((a0,a1,a2),(b0,b1,b2)) where
  a0 = cos a
```

```
  a1 = -sin a
  a2 = s * sinc (a/2) * cos (a/2)
  b0 = sin a
  b1 = cos a
  b2 = s * sinc (a/2) * sin (a/2)

sinc :: Float -> Float
sinc o
  | o ≠ 0 = sin o / o
  | otherwise = 1

qMatrix :: Float -> ((Float, Float), (Float, Float))
qMatrix a = ((1, tan (a/2)), (-tan (a/2), 1))

calcBuffer :: (Float, Float) -> (Float, Float) -> Float
calcBuffer (sMin, sMax) (aMin, aMax) = 1/6 * (av*av) * maxS + (1-cos av) * maxR
  where
    av = (aMax-aMin)/2
    maxS = max (abs sMin) (abs sMax)
    maxR = d
```

B Isabelle Implementation

```
theory SafetyZoneTwo
  imports Complex_Main
begin

no_notation Sum_Type.Plus (infixr "<+>" 65)

function enumFromTo :: "nat ⇒ nat ⇒ nat list" where
"enumFromTo b l = (if b ≤ l then b # enumFromTo (Suc b) l else [])" by auto

termination enumFromTo
  apply (relation "measure (λ(b,l). l+1 -b)")
  apply (auto)
done

fun maximum :: "('a list) ⇒ 'a :: {ord}"
  where
```

```
"maximum [] = undefined" |
"maximum (x # []) = x" |
"maximum (x # xs) = (if x > maximum xs then x else maximum xs)"

fun minimum :: "('a list) ⇒ 'a :: {ord}"
  where
"minimum [] = undefined" |
"minimum (x # []) = x" |
"minimum (x # xs) = (if x < minimum xs then x else minimum xs)"

fun concatMap :: "('a ⇒ 'b list) ⇒ 'a list ⇒ 'b list" where
"concatMap f xs = foldr (λ acc. (f x) @ acc) xs []"

fun zipWith :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a list ⇒ 'b list ⇒ 'c list" where
"zipWith f [] ys = []" |
"zipWith f xs [] = []" |
"zipWith f (x#xs) (y#ys) = f x y # zipWith f xs ys"

fun unzip :: "('a * 'b) list ⇒ 'a list * 'b list" where
"unzip xs = foldr (λ(a,b) (as,bs). (a#as,b#bs)) xs ([],[])"

type_synonym Velocity = "real"
type_synonym Distance = "real"
type_synonym Time = "real"
type_synonym BufferRadius = "real"
type_synonym Point = "(real * real)"
type_synonym SafetyZone = "(Point list * BufferRadius)"

definition brakingMeasurements :: "(Velocity * Distance) list" where
"brakingMeasurements = [(0, 0), (203.86, 113.5), (535.131, 365.0)]"

definition latency :: "Time" where
"latency = 0.06"

definition robotPoints :: "Point list" where
"robotPoints = [(−233.5,−162.5), (193.5,−162.5), (193.5,162.5), (−233.5,162.5)]"

definition l :: "nat" where
"l = 8"
```

```
definition d :: "real" where
"d = (let vLen = (λ(x,y). sqrt (x*x + y*y))
  in maximum (map vLen robotPoints))"

fun mminus :: "real * real ⇒ real * real ⇒ real * real" (infixl "<->" 6) where
  "mminus (x1,y1) (x2,y2) = (x1-x2,y1-y2)"

fun mplus :: "real * real ⇒ real * real ⇒ real * real" (infixl "<+>" 6) where
  "mplus (x1,y1) (x2,y2) = (x1+x2, y1+y2)"

 i·¿
fun mdot :: "real ⇒ real * real ⇒ real * real" (infixl "<.>" 7) where
  "mdot s (x,y) = (s*x,s*y)"

fun mmdot :: "((real * real) * (real * real)) ⇒ real * real ⇒ real * real" (infixl
     "<..>" 7) where
"mmdot ((a0,a1),(b0,b1)) (x,y) = (a0*x + a1*y, b0*x + b1*y)"

fun mmmdot :: "((real * real * real) * (real * real * real)) ⇒ real * real ⇒ real
    * real" (infixl "<..>" 7) where
"mmmdot ((a0,a1,a2),(b0,b1,b2)) (x,y) = (a0*x + a1*y + a2, b0*x + b1*y + b2)"

fun mkCandidates :: "real * real ⇒ real * real ⇒ (real * real) list" where
"mkCandidates (vMin,vMax) (wMin,wMax) = (
  let
    velos = if 0 ≥ vMin ∧ 0 ≤ vMax then [vMin,0,vMax] else [vMin,vMax];
    omega = if 0 ≥ wMin ∧ 0 ≤ wMax then [wMin,0,wMax] else [wMin,wMax]
  in concatMap (λ. concatMap (λ. [(v,w)]) omega) velos)"


function findIndexGo :: "Velocity ⇒ nat ⇒ nat ⇒ nat" where
"findIndexGo v imin imax =
  (let i = ((imax+imin) div 2) in
    if imax-imin > 1
    then if (fst (brakingMeasurements ! i)) < v
         then findIndexGo v i imax
         else findIndexGo v imin i
    else imin)" by pat_completeness auto
```

```
termination findIndexGo
  apply (relation "measure (λ(v,imin,imax). imax–imin)")
  apply (auto)
done

fun findIndex :: "Velocity ⇒ nat" where
"findIndex v = findIndexGo v 0 (length brakingMeasurements–1)"

fun ts :: "real * real ⇒ real" where
"ts (v,w) =
  (let
    vs = sqrt (v*v+d*d*w*w);
    (vm,sm) = brakingMeasurements ! (length brakingMeasurements – 1);
    (v1,s1) = brakingMeasurements ! 1
  in
    if vs ≥ vm then sm / (vm*vm*vm) * vs * vs
    else
      if vs ≤ v1 then s1 / v1
      else
        (let
          j' = findIndex vs;
          (vj', sj') = brakingMeasurements ! j';
          (vj, sj ) = brakingMeasurements ! (j'+1)
        in (sj' + ((sj–sj')/(vj–vj')) * (vs–vj')) / vs))"

fun bm :: "(real * real) ⇒ (real * real)" where
"bm (v,w) = (let ts' = ts (v,w) in ((ts'+latency)*v,(ts'+latency)*w))"

fun stepOne :: "real * real ⇒ real * real ⇒ (real * real) * (real * real)" where
"stepOne (vMin,vMax) (wMin, wMax) =
  (let
    candidates = mkCandidates (vMin,vMax) (wMin, wMax);
    (sCandidates, αCandidates) = unzip (map bm candidates);
    sMin = minimum sCandidates;
    sMax = maximum sCandidates;
    aMin = minimum αCandidates;
    aMax = maximum αCandidates
  in ((sMin,sMax), (aMin,aMax)) )"
```

```
fun sinc :: "real ⇒ real" where
"sinc o  = (if o = 0 then 1 else (sin o) / o)"

fun mkInterval :: "real * real ⇒ real * real ⇒ (real * real) list" where
"mkInterval (smin,smax) (amin,amax) =
  concatMap (λ. concatMap (λ. [(s,a)]) [amin,amax]) [smin,smax]"


fun tMatrix :: "real ⇒ real ⇒ ((real * real * real) * (real * real * real))" where
"tMatrix s a = (let
  a0 = cos a;
  a1 = −sin a;
  a2 = s * sinc (a/2) * cos (a/2);
  b0 = sin a;
  b1 = cos a;
  b2 = s * sinc (a/2) * sin (a/2) in ((a0,a1,a2),(b0,b1,b2)))"


fun qMatrix :: "real ⇒ ((real * real) * (real * real))" where
"qMatrix a = ((1, tan (a/2)), (−tan (a/2), 1))"



fun p1 :: "Point ⇒ (real * real) ⇒ Point" where
"p1 p t = t"

fun p2 :: "Point ⇒ (real * real) ⇒ Point" where
"p2 ri (s,a) = mmmdot (tMatrix s a) ri"


fun u2 :: "Point ⇒ (real * real) ⇒ Point" where
"u2 ri (s,a) = mmmdot (tMatrix (s/l) (a/l)) ri"


fun u3 :: "Point ⇒ (real * real) ⇒ Point" where
"u3 ri (s,a) = mplus ri (mmdot (qMatrix (a/l)) (0.5 <.> ((u2 ri (s,a)) <-> ri)))"
```

```
fun v :: "real ⇒ Point ⇒ (real * real) ⇒ Point" where
"v j ri (s,a) = mmmdot (tMatrix ((j*s)/l) ((j*a)/l)) (u3 ri (s,a))"

fun stepTwo :: "(real * real) ⇒ (real * real) ⇒ Point list" where
"stepTwo (smin,smax) (amin,amax) =
  (let
    interval = mkInterval (smin,smax) (amin,amax);
    p1p2v = p1 # p2 # (map v (enumFromTo 0 (l−1)))
  in concatMap (λ ∘ concatMap (λ ∘ concatMap (λ .[f ri sa]) p1p2v) robotPoints)
      interval)"


fun calcBuffer :: "real * real ⇒ real * real ⇒ real" where
"calcBuffer (sMin,sMax) (aMin,aMax) =
  (let
    av = (aMax–aMin) / 2;
    maxS = max (abs sMin) (abs sMax)
  in 1/6 * (av*av) * maxS + (1−cos av) * d)"

fun safetyZone :: "real * real ⇒ real * real ⇒ SafetyZone" where
"safetyZone vMinMax wMinMax =
  (let
    (sMinMax, aMinMax) = stepOne vMinMax wMinMax
  in (stepTwo sMinMax aMinMax, calcBuffer sMinMax aMinMax))"

lemma unzipEqualDoubleMap [simp]: "unzip xs = (map fst xs, map snd xs)"
  apply (induction xs)
  apply (auto)
done

lemma fstMap [simp]: "∀(x,y) ∈ (set xs) ∘ x ∈ set (map fst xs)" by (simp add:
    case_prod_unfold)
lemma sndMap [simp]: "∀(x,y) ∈ (set xs) ∘ y ∈ set (map snd xs)" by (simp add:
    case_prod_unfold)
lemma mapStructure [simp]: "length xs = length (map f xs)" by simp


lemma mkCandidatesResult_1 [simp] :
  "0 ≥ vMin ∧ 0 ≤ vMax ∧ 0 ≥ ωMin ∧ 0 ≤ ωMax →
```

```
    mkCandidates (vMin, vMax) (ωMin, ωMax) =
      [(vMin, ωMin), (vMin, 0), (vMin, ωMax),
       (0, ωMin), (0, 0), (0, ωMax),
       (vMax, ωMin), (vMax, 0), (vMax, ωMax)]" by simp

lemma mkCandidatesResult_2 [simp] :
  "¬(0 ≥ vMin ∧ 0 ≤ vMax) ∧ 0 ≥ ωMin ∧ 0 ≤ ωMax →
    mkCandidates (vMin, vMax) (ωMin, ωMax) =
      [(vMin, ωMin), (vMin, 0), (vMin, ωMax),
       (vMax, ωMin), (vMax, 0), (vMax, ωMax)]" by simp

lemma mkCandidatesResult_3 [simp] :
  "0 ≥ vMin ∧ 0 ≤ vMax ∧ ¬(0 ≥ ωMin ∧ 0 ≤ ωMax) →
    mkCandidates (vMin, vMax) (ωMin, ωMax) =
      [(vMin, ωMin), (vMin, ωMax),
       (0, ωMin), (0, ωMax),
       (vMax, ωMin), (vMax, ωMax)]" by simp

lemma mkCandidatesResult_4 [simp] :
  "¬(0 ≥ vMin ∧ 0 ≤ vMax) ∧ ¬(0 ≥ ωMin ∧ 0 ≤ ωMax) →
    mkCandidates (vMin, vMax) (ωMin, ωMax) =
      [(vMin, ωMin), (vMin, ωMax),
       (vMax, ωMin), (vMax, ωMax)]" by simp

lemma functorLaw_2 [simp]: "map f (map g xs) = map (λx. f (g x)) xs" by simp

lemma concatMapSimplified [simp]: "concatMap f xs = concat (map f xs)"
  apply (induction xs)
  apply (auto)
done

lemma mkLimitMovementsResult [simp]: "mkInterval (sMin, sMax) (aMin, aMax) = [
(sMin,aMin), (sMin,aMax), (sMax,aMin), (sMax,aMax)]" by simp



definition segment :: "(real * real) ⇒ (real * real) ⇒ Point set" where
"segment u v' = λ{ z. ∃α ∈ ℝ ∘ 0 ≤ α ∧ α ≤ 1 ∧ (z = (α <.> u <+> (1−α) <.> v')) λ}"
```

```
definition is_convex :: "Point set ⇒ bool" where
"is_convex K = (∀u ∈ K. ∀v ∈ K. segment u v ⊂ K)"

 convex hull is the smallest convex set
definition conv :: "Point set ⇒ Point set" where
"conv X = ∩{K ∘ is_convex K ∧ X ⊂ K}"


definition straightBrakingDistance :: "real ⇒ real" where
"straightBrakingDistance v' =
  (let
    j   = The (λx. (fst (brakingMeasurements ! (x−1))) ≤ v'
                 ∧ v' ≤ (fst (brakingMeasurements ! x)));
    (vj',sj') = brakingMeasurements ! (j−1);
    (vj,sj) = brakingMeasurements ! j;
    (v1,s1) = brakingMeasurements ! 1;
    (vm,sm) = brakingMeasurements ! (length brakingMeasurements − 1)
  in
    (if v' ≤ v1 then s1/v1∗v'
     else if v' ≥ vm then sm/(vm∗vm∗vm)∗v'∗v'∗v'
     else  sj' + (sj−sj') / (vj−vj') ∗ (v'−vj')))"

fun BMΔt :: "real ∗ real ⇒ real ∗ real" where
"BMΔt (v',ω) =
  (let
    vsHat = sqrt (v'^2+d^2∗ω^2);
    sHat = straightBrakingDistance vsHat
  in
    (sHat / vsHat + latency) <.> (v',ω) )"

definition ℌ :: "real ⇒ real ⇒ (real ∗ real) set" where
"ℌ s α = Union{ {tMatrix (x∗s) (x∗α) <.> r
  | r. r ∈ (set robotPoints)} | x. x ∈ ℝ ∧ x ≥ 0 ∧ x ≤ 1 }"

theorem mainSafetyTheorem :
  "(v',ω) ∈ ℝ × ℝ ∧ v' ≥ vMin ∧ v' ≤ vMax ∧ ω ≥ ωMin ∧ ω ≤ ωMax →
    (let
      𝔖 = conv (set (fst (safetyZone (vMin,vMax) (ωMin, ωMax))));
      (s,α) = BMΔt(v',ω)
```

```
    in
     ℌ ⊃ (ℏ s α) )"
apply clarify
apply (unfold Let_def)
apply auto
```

C Additional Proof Material

Unfolding u2 R_i (s, α) with l being the configuration variable L.

$$
\begin{aligned}
\text{u2}\ \ R_i\ (s,\alpha) = &\ \ \text{tMatrix}\ \ (s/l)\ (\alpha/l)\ \ <...>\ \ R_i \\
= &\ \texttt{let}\ (x_{R_i}, y_{R_i}) = R_i \\
&\ \texttt{in}\ \ \text{tMatrix}\ \ (s/l)\ (\alpha/l)\ \ <...>\ (x_{R_i}, y_{R_i}) \\
= &\ \texttt{let}\ (x_{R_i}, y_{R_i}) = R_i \\
&\ \texttt{in}\ ((a0, a1, a2), (b0, b1, b2))\ \ <...>\ (x_{R_i}, y_{R_i}) \\
&\quad \texttt{where} \\
&\qquad a0\ \ =\ \ \cos\ (\alpha/l) \\
&\qquad a1\ \ =\ \ -\sin\ (\alpha/l) \\
&\qquad a2\ \ =\ \ (s/l)\ *\ \text{sinc}\ ((\alpha/l)/2)\ *\ \cos\ ((\alpha/l)/2) \\
&\qquad b0\ \ =\ \ \sin\ ((\alpha/l)/l) \\
&\qquad b1\ \ =\ \ \cos\ ((\alpha/l)/l) \\
&\qquad b2\ \ =\ \ (s/l)\ *\ \text{sinc}\ ((\alpha/l)/2)\ *\ \sin\ ((\alpha/l)/2) \\
= &\ \texttt{let} \\
&\quad (x_{R_i}, y_{R_i})\ \ =\ \ R_i \\
&\quad xv\ \ = \cos\ (\alpha/l)\ *\ x_{R_i}\ +\ (-\sin\ (\alpha/l))\ *\ y_{R_i}\ +\ ((s/l)\ *\ \text{sinc}\ ((\alpha/l)/2)\ *\ \cos\ ((\alpha/l)/2)) \\
&\quad yv\ \ = \sin\ (\alpha/l)\ *\ x_{R_i}\ +\ (\cos\ (\alpha/l))\ *\ y_{R_i}\ +\ ((s/l)\ *\ \text{sinc}\ ((\alpha/l)/2)\ *\ \sin\ ((\alpha/l)/2)) \\
&\ \texttt{in}\ (xv, yv)
\end{aligned}
$$

Next, unfolding $U^2_{i,s,\alpha}$.

$$
\begin{aligned}
U^2_{i,s,\alpha} &= T(\frac{s}{L}, \frac{\alpha}{L}) R_i \\
&= \begin{pmatrix} \cos\frac{\alpha}{L} & -\sin\frac{\alpha}{L} & \frac{s}{L}\cdot\text{sinc}\frac{\frac{\alpha}{L}}{2}\cdot\cos\frac{\frac{\alpha}{L}}{2} \\ \sin\frac{\alpha}{L} & \cos\frac{\alpha}{L} & \frac{s}{L}\cdot\text{sinc}\frac{\frac{\alpha}{L}}{2}\cdot\sin\frac{\frac{\alpha}{L}}{2} \end{pmatrix} \begin{pmatrix} x_{R_i} \\ y_{R_i} \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos\frac{\alpha}{L}\cdot x_{R_i} + (-\sin\frac{\alpha}{L})\cdot y_{R_i} + \frac{s}{L}\cdot\text{sinc}\frac{\frac{\alpha}{L}}{2}\cdot\cos\frac{\frac{\alpha}{L}}{2} \\ \sin\frac{\alpha}{L}\cdot x_{R_i} + \cos\frac{\alpha}{L}\cdot y_{R_i} + \frac{s}{L}\cdot\text{sinc}\frac{\frac{\alpha}{L}}{2}\cdot\sin\frac{\frac{\alpha}{L}}{2} \end{pmatrix}
\end{aligned}
$$

Hence, u2 R_i $(s, \alpha) \cong U^2_{i,s,\alpha}$.

□

Unfolding u3 R_i (s, α).

$$
\begin{aligned}
\texttt{u3}\ \ u1@ri\ sa@(s,\alpha) &= u1\ \texttt{<+>}\ \ \texttt{qMatrix}\ \ (\alpha/l)\ \texttt{<..>}\ (0.5\ \texttt{<.>}\ (\ \texttt{u2}\ \ R_i\ sa\ \texttt{<->}\ u1)) \\
&= R_i\ \texttt{<+>}\ \ \texttt{qMatrix}\ \ (\alpha/l)\ \texttt{<..>}\ (0.5\ \texttt{<.>}\ (\ \texttt{u2}\ \ R_i\ sa\ \texttt{<->}\ R_i)) \\
&= \texttt{let}\ (u2x, u2y)\ =\ \texttt{u2}\ \ R_i\ sa \\
&\quad \texttt{in}\ R_i\ \texttt{<+>}\ \ \texttt{qMatrix}\ \ (\alpha/l)\ \texttt{<..>}\ (0.5\ \texttt{<.>}\ ((u2x, u2y)\ \texttt{<->}\ R_i)) \\
&= \texttt{let} \\
&\quad\ (u2x, u2y)\ =\ \texttt{u2}\ \ R_i\ sa \\
&\quad\ (x_{R_i}, y_{R_i})\ =\ R_i \\
&\quad \texttt{in}\ (x_{R_i}, y_{R_i})\ \texttt{<+>}\ \ \texttt{qMatrix}\ \ (\alpha/l)\ \texttt{<..>}\ (0.5\ \texttt{<.>}\ ((u2x, u2y)\ \texttt{<->}\ (x_{R_i}, y_{R_i}))) \\
&= \texttt{let} \\
&\quad\ (u2x, u2y)\ =\ \texttt{u2}\ \ R_i\ sa \\
&\quad\ (x_{R_i}, y_{R_i})\ =\ R_i \\
&\quad \texttt{in}\ (x_{R_i}, y_{R_i})\ \texttt{<+>}\ \ \texttt{qMatrix}\ \ (\alpha/l)\ \texttt{<..>}\ (0.5\ \texttt{<.>}\ (u2x - x_{R_i}, u2y - y_{R_i})) \\
&= \texttt{let} \\
&\quad\ (u2x, u2y)\ =\ \texttt{u2}\ \ R_i\ sa \\
&\quad\ (x_{R_i}, y_{R_i})\ =\ R_i \\
&\quad \texttt{in}\ (x_{R_i}, y_{R_i})\ \texttt{<+>}\ \ \texttt{qMatrix}\ \ (\alpha/l)\ \texttt{<..>}\ (0.5 * (u2x - x_{R_i}), 0.5 * (u2y - y_{R_i})) \\
&= \texttt{let} \\
&\quad\ (u2x, u2y)\ =\ \texttt{u2}\ \ R_i\ sa \\
&\quad\ (x_{R_i}, y_{R_i})\ =\ R_i \\
&\quad \texttt{in}\ (x_{R_i}, y_{R_i})\ \texttt{<+>}\ ((1, \tan\ ((\alpha/l)/2)), (-\tan\ ((\alpha/l)/2), 1))\ \texttt{<..>}\ (0.5 * (u2x - x_{R_i}), 0.5 * (u2y - y_{R_i})) \\
&= \texttt{let} \\
&\quad\ (u2x, u2y)\ =\ \texttt{u2}\ \ R_i\ sa \\
&\quad\ (x_{R_i}, y_{R_i})\ =\ R_i \\
&\quad\ x\ =\ 1 * (0.5 * (u2x - x_{R_i})) + (\tan\ ((\alpha/l)/2)) * (0.5 * (u2y - y_{R_i})) \\
&\quad\ y\ =\ (-\tan\ ((\alpha/l)/2)) * (0.5 * (u2x - x_{R_i})) + 1 * (0.5 * (u2y - y_{R_i})) \\
&\quad \texttt{in}\ (x_{R_i}, y_{R_i})\ \texttt{<+>}\ (x, y) \\
&= \texttt{let} \\
&\quad\ (u2x, u2y)\ =\ \texttt{u2}\ \ R_i\ sa \\
&\quad\ (x_{R_i}, y_{R_i})\ =\ R_i \\
&\quad\ x\ =\ 1 * (0.5 * (u2x - x_{R_i})) + (\tan\ ((\alpha/l)/2)) * (0.5 * (u2y - y_{R_i})) \\
&\quad\ y\ =\ (-\tan\ ((\alpha/l)/2)) * (0.5 * (u2x - x_{R_i})) + 1 * (0.5 * (u2y - y_{R_i})) \\
&\quad \texttt{in}\ (x_{R_i} + x, y_{R_i} + y)
\end{aligned}
$$

Next,unfolding $U^3_{i,s,\alpha}$.

$$
\begin{aligned}
U^3_{i,s,\alpha} &= U^1_{i,s,\alpha} + Q(\frac{\alpha}{L})\frac{1}{2}(U^2_{i,s,\alpha} - U^1_{i,s,\alpha}) \\
&= R_i + Q(\frac{\alpha}{L})\frac{1}{2}(U^2_{i,s,\alpha} - R_i) \\
&= \begin{pmatrix} x_{R_i} \\ y_{R_i} \end{pmatrix} + \begin{pmatrix} 1 & \tan\frac{\alpha}{2L} \\ -\tan\frac{\alpha}{2L} & 1 \end{pmatrix} \frac{1}{2} \left(\begin{pmatrix} x_{U^2_{i,s,\alpha}} \\ y_{U^2_{i,s,\alpha}} \end{pmatrix} - \begin{pmatrix} x_{R_i} \\ y_{R_i} \end{pmatrix} \right) \\
&= \begin{pmatrix} x_{R_i} \\ y_{R_i} \end{pmatrix} + \begin{pmatrix} 1 & \tan\frac{\alpha}{2L} \\ -\tan\frac{\alpha}{2L} & 1 \end{pmatrix} \frac{1}{2} \begin{pmatrix} x_{U^2_{i,s,\alpha}} - x_{R_i} \\ y_{U^2_{i,s,\alpha}} - y_{R_i} \end{pmatrix} \\
&= \begin{pmatrix} x_{R_i} \\ y_{R_i} \end{pmatrix} + \begin{pmatrix} 1xv_i + (\tan\frac{\alpha}{2L})yv_i \\ (-\tan\frac{\alpha}{2L})xv_i + 1yv_i \end{pmatrix} \\
&= \begin{pmatrix} x_{R_i} + 1xv_i + (\tan\frac{\alpha}{2L})yv_i \\ y_{R_i} + (-\tan\frac{\alpha}{2L})xv_i + 1yv_i \end{pmatrix}
\end{aligned}
$$

with

$$
\begin{aligned}
R_i &= \begin{pmatrix} x_{R_i} \\ y_{R_i} \end{pmatrix} \\
xv_i &= \frac{1}{2}(x_{U^2_{i,s,\alpha}} - x_{R_i}) \\
yv_i &= \frac{1}{2}(y_{U^2_{i,s,\alpha}} - y_{R_i})
\end{aligned}
$$

Thus, u3 $R_i\ (s,\alpha) \cong U^3_{i,s,\alpha}$.

□

Next, one can unfold $\mathrm{v}\ \ j\ R_i\ (s,\alpha)$.

$$
\begin{aligned}
\mathrm{v}\ \ j\ R_i\ sa@(s,\alpha) &= \ \ \mathrm{tMatrix}\ \ ((j*s)/l)\ ((j*\alpha)/l)\ \ <\ldots>\ \ \mathrm{u3}\ \ \ R_i\ sa \\
&= \ \mathtt{let} \\
&\quad\ (x_{U^3_{i,s,alpha}}, y_{U^3_{i,s,alpha}}) \ = \ \ \mathrm{u3}\ \ \ R_i\ sa \\
&\quad \mathtt{in}\ \ \mathrm{tMatrix}\ \ ((j*s)/l)\ ((j*\alpha)/l)\ \ <\ldots>\ (x_{U^3_{i,s,alpha}}, y_{U^3_{i,s,alpha}}) \\
&= \ \mathtt{let} \\
&\quad\ (x_{U^3_{i,s,alpha}}, y_{U^3_{i,s,alpha}}) \ = \ \ \mathrm{u3}\ \ \ R_i\ sa \\
&\quad \mathtt{in}\ ((a0,a1,a2),(b0,b1,b2))\ \ <\ldots>\ (x_{U^3_{i,s,alpha}}, y_{U^3_{i,s,alpha}}) \\
&\qquad \mathtt{where} \\
&\qquad\quad a0 \ = \ \cos\ ((j*\alpha)/l) \\
&\qquad\quad a1 \ = \ -\sin\ ((j*\alpha)/l) \\
&\qquad\quad a2 \ = \ ((j*s)/l)\ *\ \mathrm{sinc}\ (((j*\alpha)/l)/2)\ *\ \cos\ (((j*\alpha)/l)/2) \\
&\qquad\quad b0 \ = \ \sin\ (((j*\alpha)/l)/l) \\
&\qquad\quad b1 \ = \ \cos\ (((j*\alpha)/l)/l) \\
&\qquad\quad b2 \ = \ ((j*s)/l)\ *\ \mathrm{sinc}\ (((j*\alpha)/l)/2)\ *\ \sin\ (((j*\alpha)/l)/2) \\
&= \ \mathtt{let} \\
&\quad\ (x_{U^3_{i,s,alpha}}, y_{U^3_{i,s,alpha}}) \ = \ \ \mathrm{u3}\ \ \ R_i\ sa \\
&\quad\ xv \ = \ (\cos\ ((j*\alpha)/l)) * x_{U^3_{i,s,alpha}} + (-\sin\ ((j*\alpha)/l)) * y_{U^3_{i,s,alpha}} \\
&\qquad\qquad + (((j*s)/l)\ *\ \mathrm{sinc}\ (((j*\alpha)/l)/2)\ *\ \cos\ (((j*\alpha)/l)/2)) \\
&\quad\ yv \ = \ (\sin\ (((j*\alpha)/l)/l)) * x_{U^3_{i,s,alpha}} + (\cos\ (((j*\alpha)/l)/l)) * y_{U^3_{i,s,alpha}} \\
&\qquad\qquad + (((j*s)/l)\ *\ \mathrm{sinc}\ (((j*\alpha)/l)/2)\ *\ \sin\ (((j*\alpha)/l)/2)) \\
&\quad \mathtt{in}\ (xv, yv)
\end{aligned}
$$

After that, one can unfold $V^j_{i,s,\alpha}$.

$$
\begin{aligned}
V^j_{i,s,\alpha} &= T\Big(\frac{j\cdot s}{L}, \frac{j\cdot\alpha}{L}\Big)\cdot U^3_{i,s,\alpha} \\
&= \begin{pmatrix} \cos\frac{j\cdot\alpha}{L} & -\sin\frac{j\cdot\alpha}{L} & \frac{j\cdot s}{L}\cdot\mathrm{sinc}\frac{\frac{j\cdot\alpha}{L}}{2}\cdot\cos\frac{\frac{j\cdot\alpha}{L}}{2} \\ \sin\frac{j\cdot\alpha}{L} & \cos\frac{j\cdot\alpha}{L} & \frac{j\cdot s}{L}\cdot\mathrm{sinc}\frac{\frac{j\cdot\alpha}{L}}{2}\cdot\sin\frac{\frac{j\cdot\alpha}{L}}{2} \end{pmatrix} \cdot \begin{pmatrix} x_{U^3_{i,s,\alpha}} \\ y_{U^3_{i,s,\alpha}} \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} (\cos\frac{j\cdot\alpha}{L})\ x_{U^3_{i,s,\alpha}} + (-\sin\frac{j\cdot\alpha}{L})\ y_{U^3_{i,s,\alpha}} + (\frac{j\cdot s}{L}\cdot\mathrm{sinc}\frac{\frac{j\cdot\alpha}{L}}{2}\cdot\cos\frac{\frac{j\cdot\alpha}{L}}{2}) \\ (\sin\frac{j\cdot\alpha}{L})\ x_{U^3_{i,s,\alpha}} + (\cos\frac{j\cdot\alpha}{L})\ y_{U^3_{i,s,\alpha}} + (\frac{j\cdot s}{L}\cdot\mathrm{sinc}\frac{\frac{j\cdot\alpha}{L}}{2}\cdot\sin\frac{\frac{j\cdot\alpha}{L}}{2}) \end{pmatrix}
\end{aligned}
$$

Therefore, $\mathrm{v}\ \ j\ R_i\ (s,\alpha) \cong V^j_{i,s,\alpha}$ follows.

□