COMP5314
MSc Computational Intelligence and Robotics Project
A Functional Approach to Fuzzy Logic

Dave Tapley
`cir06dt@dmu.ac.uk`

September 21, 2007

Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Scope for work

As to be reviewed in Section 2.3 there are several existing papers on the use of the functional programming paradigm in fuzzy logic applications. However it is the opinion of the author that for several factors there is still a need for further research into the area, these include:

- The lack of recent [1] published work on functional programming and fuzzy logic, despite a rapid growth in interest in the field of functional programming. The growth is documented by Simon Peyton-Jones during a talk on Haskell given in 2007 [1].
- Specifically the functional language Haskell has seen many developments since the previous work was undertaken, leveraging these may lead to further gains in the area.
- Whilst the articles reviewed provide some demonstrations of their use they do not investigate in great depth the potential for the use of their implementations as heavyweight modules. By this we mean a providing a fuzzy logic component which could be used in a real application, without requiring knowledge of the underlying code.

1.1.2 Need for modules

A project goal is to provide reusable Haskell modules to allow other users to easily take advantage of fuzzy logic in their projects and to do so without unnecessary reimplementation of fuzzy logic code. Satisfactory realisation of this will increase the dissemination potential for the work undertaken in this project.

There are several factors which need to be addressed for the work undertaken in this project to be considered suitable to use as fuzzy logic modules:

1. It is presupposed that any module of code (that is any single file) would used in a read only fashion in projects which use it. We expect that a future user would bring the functions provided by this work in to scope with an import statement and evaluate them within their own modules (defined in their own files). We do not want a situation where the user be required to modify any of the code provided in this report.
2. With this restriction is in place we must ensure a user is able to configure the system to meet their own needs to the maximum degree possible. As such we must find an elegant method for specifying parameters externally, such that the implementations given in this project can adjust their behaviour accordingly.
3. Finally if the work given here does not provide all the functionality required by a user; but does cover a portion of it then we expect them to still be able to use the modules. That is we must provide the means to extend the modules provided here in a fashion which does not require reimplementation of the given code.

1.2 Aims

The novelty of this project is centred in its use of the functional programming paradigm and so the core aim shall be the demonstration that firstly it is possible to build a fuzzy system in a functional language and secondly that there are features specific to the paradigm which simplify the process.

Noe we now look at the different aspects of this central aim:

1. A literature review shall be conducted to identify existing fuzzy logic research which has been undertaken using functional languages. We show how the development of a fuzzy system was possible and explore what advantages these works demonstrated which we may exploit or develop further.

[1]No post 2000 articles specific to the implementation of fuzzy logic in a functional programming paradigm could be found.

2. An implication of this is that we aim to develop a system which is more sophisticated than the ones identified during the literature review. We propose two ways in which we can develop a more sophisticated system:

   - We can add new features, extending the variety of tasks it can perform.
   - Alternatively we can approach the same problem in a different way and develop a system which performs similarly but which has merits the original work does not.

   In this project we shall seek a balance between these two, allowing us to demonstrate the additional potential the functional paradigm has to offer to existing work whilst exploring other entirely new approaches.

3. We just identified a balance between two ways in which we can demonstrate the benefits of the functional paradigm: Firstly building a more feature full system and secondly re-approaching existing ones. For the former we aim will be to provide an argument against the conception that the functional approach is restricted to the research domain and that it becomes counter-productive as system size grows. For the second we adopt an opposite angle and aim to demonstrate that there are properties which make the research of new approaches more appealing and simpler in the functional paradigm.

1.3 Objectives

Our primary objective has been identified as the development of fuzzy logic modules for the Haskell programming language which satisfy the requirements discussed in Section 1.1.2. To be more rigorous about this we shall state that we will present a system in Haskell which demonstrates a problem being solved using a fuzzy logic approach; this divides the system in to two components:

A fuzzy system in which the specific problem to be solved is defined using fuzzy sets and rules.

Supporting modules which shall provide all the functionality to perform inferencing with these sets and rules and produce a meaningful (defuzzified) output from the system.

A second objective is to demonstrate the reuse of existing code, this comprises of two uses: Internally within the report we shall see how functionality may be derived from existing functions rather than requiring replication and minor modification of existing ones. Secondly the ease with which the framework and function definitions given within this report may be used if the work is extended.

We can extend this to encapsulate any code presented in existing research work (to be covered in the literature review). It is a sensible objective to see if we can utilise this directly in this work by using function definitions taken verbatim or with as little modification as possible.

Chapter 2

Literature review

2.1 Summary of chapter

In a review of literature relevant to the project we firstly cover fuzzy logic. As the subject is core to project this shall provide the uninformed reader with a broad overview of the topic and provide a deeper detail of those areas specific to the project goals. Initially a review of functional programming shall be provided, again this is intended to provide sufficient depth to introduce a non functional programmer to the topic and cite sources where further details can be found. It is not the intention of this section to present an argument for the use of functional programming, however many of aspects covered will form components of the author's case for functional programming later in Section 3.

2.2 Fuzzy logic

In this first section of the literature review the author shall present an overview of fuzzy logic as described by historical and authoritative publications in the field. It is the author's intention that the review presented here shall detail only with theoretical and notational aspects and disregard implementations or applications.

It is common [2, 3, 4, 5] for introductory books on the subject of fuzzy logic present first the idea of classical set theory[1] and then fuzzy set theory as generalisation of it as identified by Kaufmann [6]. Whilst this is a natural way to introduce the concept R. Seising observed that "the genesis of fuzzy sets is not a story of basic research in set theory [..] but it is a story of fundamental research of a mathematical oriented electrical engineer" [7]. With a basis in set theory established it is then shown how a membership function (commonly denoted by μ) can be used to define a fuzzy set and common set operations such as union ($\cup$), intersection ($\cap$) and compliment ($\bar{\ }$) are extended to create a fuzzy logic.

Accordingly the author shall present a summary of their findings below adhering to this order, indicating where reviewed publications present different or conflicting ideas and indicating those which are most integral to the project itself.

2.2.1 Fuzzy sets

The origin of classical or *crisp* set theory is widely accredited to Georg Cantor [8] and has its roots in the notion of *inclusion* which denotes that an object may or may not be in a set. The set of all objects is called the *universal set.* Klir [4] highlights the common notation (shown in (2.1)) which defines a set A to contain elements x from the universal set only if they satisfy some properties $P_1, P_2, \cdots, P_n$.

$$A = \{x \mid x \text{ has properties } P_1, P_2, \cdots, P_n\} \tag{2.1}$$

Membership to a set is also determined by a function which maps an object from the universal set to an indicator of of membership or non-membership, this indicator is referred to in [3] as a *valuation set*, in crisp logic is the set $\{0, 1\}$. The author has observed this function referred to under many names, the most complete list given by Mendel [2]: membership, characteristic, discrimination or indicator function. Concise expression of the characteristic function is given by (2.2).

$$\mu_A(x) = \begin{cases} 1 & \text{iff} \quad x \in A, \\ 0 & \text{iff} \quad x \notin A. \end{cases} \tag{2.2}$$

By allowing the valuation set to be the interval of real numbers $[0, 1]$ a *fuzzy set* is obtained [9]. In allowing the *grade* of membership to be any value in this range, rather than just 0 or 1[2] we have moved from a bivalent set theory to a multivalent one. Bart Kosko argues that this is a very important change in thinking in his book *Fuzzy Thinking*[10] where he makes statements such as "Bivalence trades accuracy for simplicity" and "Fuzziness or multivalance holds everywhere in between the corners. It includes the corners as special cases". The corners mentioned in the latter statement here refer to zero and one, at either end of the fuzzy valuation set.

It should be noted that in Zadeh's 1965 paper[9] which is accredited as the first article on fuzzy sets[7] he uses the notation $f_A(x)$ to denote the "grade of membership" for x, in fuzzy set A in a universal set X ($\forall x \in X$). The author observes that this notation appears not to have stuck in

[1]Where an object must either be present in a set or not present in it.

[2]Indicating total absence or membership to a set respectively.

literature and the notation used for the characteristic function in strict sets ($\mu_A(x)$) is now used exclusively. Further to avoid ambiguity the author shall use the term *characteristic* function to refer to *crisp* sets (where the valuation set is $\{0, 1\}$), and the term *membership* function to refer to *fuzzy* sets (where the valuation set is $[0, 1]$).

Zadeh proposed another notation for fuzzy sets in a 1972 paper[11] which is presented below. It should be noted that by this time he also had moved to the $\mu_A(x)$ notation for a membership function. The new notation provided a mechanism to express both *finite* (and equivalently *discrete*) and *continuous* (and equivalently *geometric*) fuzzy sets both of which are covered now:

Finite universal set notation is an extension of the $A = \{x_1, x_2, \cdots, x_n\}$ style notation in crisp sets, except here just the presence of a member of the universal set in the list is not sufficient; thus we prefix it with its degree of membership and then a forward slash. As with the crisp notation if a member of the universal set is not present in the list then it is entirely *not* a member of the set (and thus has a degree of membership equal to zero). The notation is given formally in equation 2.3.

$$A = \mu_A(x_1)/x_1 + \cdots + \mu_A(x_n)/x_n = \sum_{i=1}^{n} \mu_A(x_i)/x_i \tag{2.3}$$

An example of a fuzzy set (indicating membership to the set tallness) defined over a finite universal set (we shall use a set of people) along with the corresponding notation and usage is given below.

- The universal set of people:

$$X = \{\text{Dave}, \text{Jenny}, \text{Simon}\}$$

- A fuzzy set tallness defined upon it:

$$T = \{0.9/\text{Dave}, 0.6/\text{Jenny}, 0.75/\text{Simon}\}$$

- Obtaining the degree of membership of domain values in the fuzzy set T.

$$\begin{aligned} \mu_T(\text{Dave}) &= 0.9 \\ \mu_T(\text{Jenny}) &= 0.6 \\ \mu_T(\text{Simon}) &= 0.75 \end{aligned}$$

Continuous universal set notation is used if the universe (X) is not finite. The summation sign of equation 2.3 is replaced with the integral sign to give equation 2.4. The most commonly seen continuous universe of discourse is the set of real numbers or a specific sub set of them.

$$A = \int_X \mu_A(x)/x \tag{2.4}$$

One may translate the tallness example used previously into one defined on a continuous universe, a natural choice for this would be the height in centimeters of the person. As initially defined by Zadeh in Fuzzy Sets [9] a membership function associates each point in X with a real number in the interval $[0, 1]$, he then states in a footnote that: “More generally, the domain of definition of $f_A(x)$ may be restricted to a sub-set of X”. In the context of our example then we shall restrict the domain of definition of our fuzzy set tallness to a set of real numbers indicating height between 100 and 200 cm. Because our domain is continuous we must provide a *continuous* membership function. *Parametric* functions are commonly used because they allow the fuzzy set to be easily visualised on a graph and provide intuitive parameters. One function identified by Garibaldi and John in [12] is the right shoulder membership function who takes two parameters which divide the domain into three regions. The degree of membership for a value in the domain is then established thus:

- The degree of membership for domain values before the first parameter are zero.

- The domain values between the first and second parameter are linearly increasing with their proximity to the second parameter.
- Lastly those domain values after the third parameter have a membership degree of one.

For our example let us consider any one who is under 150 cm heigh to be not in the set tall and anyone above 175 cm to be entirely a member of the set tall, this gives us the two parameters for our right shoulder membership function: 150 and 175. Using this a continuous domain version of the example usage is presented:

- The universal set of heights:

$$X = \{x | x \in \mathbb{R}, 100 \leqslant x \leqslant 200\}$$

- A fuzzy set tallness defined upon it:

$$\mu_T(x) = \begin{cases} 0 & x < 150, \\ \frac{x-150}{25} & 150 \leqslant x \leqslant 175, \\ 1 & x > 175. \end{cases}$$

- Obtaining the degree of membership of some domain values in the fuzzy set T.

$$\begin{aligned} \mu_T(125) &= 0 \\ \mu_T(160) &= \frac{(160-150)}{25} = 0.4 \\ \mu_T(180) &= 1 \end{aligned}$$

As hinted when noting that continuous membership functions are often derived from geometric models being able to visualise the *shape* of a membership function across the domain on which it is defined can greatly aid intuition of the nature of a fuzzy set. To demonstrate this a graph of the tallness fuzzy set is given in Figure 2.1, from this it is easy to see where the term right shoulder is obtained from.

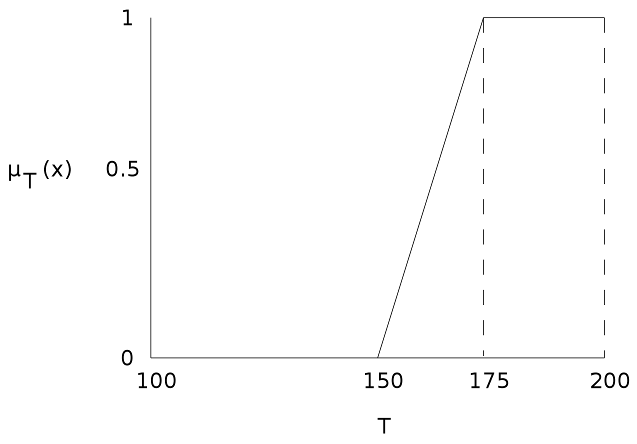

Figure 2.1: Example of a right shoulder membership function

A wide range of geometric functions make appropriate membership functions such as triangles, trapezoids and Gaussian bell curves.

2.2.2 Operations fuzzy sets

In [4] Klir and Folger present a connection between fuzzy sets and *infinite-valued logic*[3] with the statement “the membership grades $\mu_A(x)$ for $x \in X$ by which a fuzzy set A on the universal set X is

[3]Also identified in [4] as the *standard Lukasiewicz logic* L_1.

defined can be interpreted as the truth values of the proposition "x is a member of set A"". Further they state "the truth values for all $x \in X$ of any proposition "x is P" [..] where P is a vague (fuzzy) predicate [..] can be interpreted as the membership degrees $\mu_P(x)$".

This notion is covered by Zadeh under the title "Approximate reasoning" in [13]. In discussing assertions of the form "u is A ("as in Harry is tall,")" he notes how unlike the case that A is a *crisp* subset of U (e.g. all people are tall or not tall) if A is a *fuzzy* subset of U then the term "u is a member of A" is meaningless in its usual mathematical sense. He goes on to demonstrate how the concept of a *linguistic variable* forms a framework in which inferencing may lead to approximate reasoning and thus a fuzzy logic.

Because an isomorphism exists between set theory and propositional[4] logic we may derive a fuzzy logic from fuzzy set theory. In establishing a logic one combines propositions using operations such as the following[2]:

- Conjunction, denoted by $\wedge$
- Disjunction, denoted by $\vee$
- Negation, denoted by $\neg$ or $\sim$ as used by Mendel.
- Implication, denoted by $\rightarrow$ or $\Rightarrow$

To construct a fuzzy logic it is necessary to provide definitions of how these operations affect fuzzy sets when they are applied to them.

Conjunction of two propositions is the formal term for the phrase "and", implying that the conjunction is only *true* if *both* the comprising propositions are true. It follows from the isomorphism between set theory and propositional logic that conjunction is equivalent to the *intersection* operation and one such method is given by (2.5), proposed by Zadeh in [9].

$$\forall x \in X, \mu_{A \cup B}(x) = \min(\mu_A(x), \mu_B(x)) \tag{2.5}$$

In an infinite valued logic we would expect that the degree of membership (or truth as defined by Klir and Folger) to be no more than the minimum membership in either of the fuzzy sets which comprise the conjunction. This requirement was defined formally by Bellman and Giertz alongside six others in [14] where they also prove that Zadeh's min operator is the only one which meets their six requirements.

In [4] Klir and Folger give six axioms, shown below, of which the former four they call the *axiomatic skeleton for fuzzy set unions* and the latter two "desirable in certain applications". They are almost entirely equivalent to those defined by Bellman and Giertz.

Axiom i1. $i(1,1) = 1; i(0,1) = i(1,0) = i(0,0) = 0$; that is, i behaves as the classical intersection with crisp sets (*boundary condition*).

Axiom i2. $i(a,b) = i(b,a)$; this is, i is *commutative.*

Axiom i3. If $a \leqslant a'$ and $b \leqslant b'$, then $i(a,b) \leqslant i(a',b')$; that is, i is *monotonic.*

Axiom i4. $i(i(a,b),c) = i(a,i(b,c))$; that is, i is *associative.*

Axiom i5. i is a *continuous* function.

Axiom i6. $i(a,a) = a$; that is, i is *idempotent.*

Following the enumeration of these six axioms they present a table of six classes of intersection operations in addition to Zadeh's min operation.

The first four of these axioms describe a binary operation called triangular norm or t-norm, usually denoted by the symbol $\star$. In the formal definition of a t-norm the identity element is 1.

Disjunction is equivalent to the "or" statement and is isomorphic with the *union* operation in set theory. As with conjunction Zadeh's proposed method given in [9] is the only operator to satisfy the Bellman and Giertz conditions and is given by (2.6).

$$\forall x \in X, \mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x)) \tag{2.6}$$

In [4] Klir and Folger again give four required and two desirable axioms for a fuzzy union operation along with a table showing a further six classes of union operations in addition to Zadeh's max operation. The author gives the definitions now:

Axiom u1. $u(0,0) = 0; u(0,1) = u(1,0) = u(1,1) = 1$; that is, u behaves as the classical union with crisp sets (*boundary condition*).

Axiom u2. $u(a,b) = u(b,a)$; this is, u is *commutative.*

Axiom u3. If $a \leqslant a'$ and $b \leqslant b'$, then $i(a,b) \leqslant u(a',b')$; that is, u is *monotonic.*

Axiom u4. $u(u(a,b),c) = u(a,u(b,c))$; that is, u is *associative.*

Axiom u5. u is a *continuous* function.

Axiom u6. $u(a,a) = a$; that is, u is *idempotent.*

The first four of these axioms describe a binary operation called triangular conorm, t-conorm or s-norm, usually denoted by the symbol $\oplus$. In the formal definition of a t-conorm the identity element is 0.

Negation of a proposition described by a fuzzy set (for example transforming the fuzzy set "tall" into "not tall") is isomorphic to taking the *compliment* of the set, given by Zadeh in [9] as

$$\forall x \in X, \mu_{\bar{A}}(x) = 1 - \mu_A(x) \tag{2.7}$$

Whilst this appears to be a trivial function Dubois and Prade discuss in [3] that it unlike conjunction and disjunction (and equivalently intersection and union) Zadeh's operation (2.7) is not the only operation which satisfies the conditions set out by Bellman and Giertz in [14]. Following this an alternative known as the λ compliment was proposed by Sugeno [15] and is given by (2.8).

$$\mu_{\bar{A}^\lambda}(x) = \frac{1 - \mu_A(x))}{(1_\lambda \mu_A(X)}, \lambda \in (-1, \infty) \tag{2.8}$$

In their discussion of the fuzzy compliment operation in [4] Klir and Folger do not mention Bellman and Giertz's conditions and instead present the following two axiomatic *requirements*:

Axiom c1. $c(0) = 1$ and $c(1) = 0$, that is, c behaves as the ordinary compliment for crisp sets (*boundary conditions*).

Axiom c2. For all $a, b \in [0,1]$, if $a < b$, then $c(a) \geqslant c(b)$, that is c is *monotonic nonincreasing.*

They also give two *desirable* axioms presented thus:

Axiom c3. c is a continuous function.

Axiom c4. c is *involutive*, which means that $c(c(a)) = a$ for all $a \in [0,1]$.

Both the Zadeh and Sugeno compliments satisfy all four axioms c[1..4] and Klir and Folger present (2.9) as another approach to the fuzzy compliment called the *Yager class.* The notation has been adapted to be comparable with (2.7, 2.8).

$$\mu_{\bar{A}^w}(x) = (1 - x^w)^{\frac{1}{w}}, w \in (0, \infty) \tag{2.9}$$

Implication is the final operation covered. Formally it is called *material implication* and it allows the construct of *if P then Q* style linguistic statements. We define this operation as adhering to to truth table presented in Figure 2.1. It is common in literature for the proposition P to be called the *antecedent* and the proposition Q the *consequence.* It should be noted that this is different from the *equivalence* relationship denoted by $\leftrightarrow$ in that a false antecedent does not imply a false consequent.

In [2] Mendel shows concisely how we may construct *tautologies*[4] with identical truth tables (and therefore logically equivalent) to implication using the three operations already introduced. Two

[4]Defined as "propositions which are formed by combining other propositions".

P	Q	$P \to Q$
T	T	T
T	F	F
F	T	T
F	F	T

Table 2.1: Truth table for implication

examples of these are given in 2.10 and 2.11 which we can see are equivalent by applying De Morgan's laws.

$$\neg[P \wedge (\neg Q)] \tag{2.10}$$

$$(\neg P) \vee Q \tag{2.11}$$

Observing Zadeh's fuzzy logic operations for $\wedge$, $\vee$ and $\neg$ as already covered we can generate implication operations for a fuzzy logic by substitution, this gives those presented in 2.12 and 2.13 where the truth value for P is given by $\mu_A(x)$ and for Q is given by $\mu_B(y)$.

$$\mu_{A\to B}(x, y) = 1 - \min[\mu_A(x), 1 - \mu_B(y)] \tag{2.12}$$

$$\mu_{A\to B}(x, y) = \max[1 - \mu_A(x), \mu_B(y)] \tag{2.13}$$

Having covered this material Mendel provides an example in [2] which demonstrates how the use of either of these implementations in association with *generalised modus ponens* causes the introduction of a undesirable constant in the output.

Two solutions to this were indirectly obtained in pursuit of implication operations which simplified computation, they were 2.14 proposed by Mamdani in [16] and 2.15 proposed by Larsen [17].

$$\mu_{A\to B}(x, y) = \min[\mu_A(x), \mu_B(y)] \tag{2.14}$$

$$\mu_{P\to Q}(x, y) = \mu_A(x)\mu_B(y) \tag{2.15}$$

It is noted that these two solutions do not ad here to the standard definition of implication from crisp logic as given in table 2.1 and also that 2.14 and 2.15 are collectively referred to as *Mamdani implication* defined with a t-norm operator ($\star$) as given in 2.16.

$$\mu_{A\to B}(x, y) = \mu_A(x) \star \mu_B(y) \tag{2.16}$$

2.2.3 Inferencing with fuzzy sets

Having identified these isomorphic operations we now see how the crisp logic rule of modus ponens may be extended to apply in a fuzzy logic. To indicate the deviation from the crisp logic interpretation Zadeh introduces the term *generalized modus ponens*, described in [18] thus: "The compositional rule of inference leads to a generalized modus ponens, which may be viewed as an extension of the familiar rule of inference: If A is true and A implies B, then B is true."

In [2] Mendel presents generalised modus ponens in the following way:

Premise: x is A*
Implication: IF x is A THEN y is B
Consequence: y is B*

He then observes the difference between this and classical crisp modus ponens as "fuzzy set A* is not necessarily the same as rule antecedent fuzzy set A, and fuzzy set B* is not necessarily the same as rule consequent B".

The literature outlines two approaches to implementing generalised modus ponens, both of which are addressed now:

1. By considering the generalised version to be isomorphic to the classic version one obtains a resultant fuzzy set B* by taking the supremum membership value across the domain of x of a conjunction between fuzzy set A* and the fuzzy relation between A and B. The former conjunction and latter fuzzy relation are given by some conjunction and implication operators respectively such as those covered previously.

   Mendel [2] describes this as a sup-star composition, the name coming from the fact one takes the supremum and star to indicate a t-norm function for the conjunction. It is given in equation 2.17.

   $$\mu_{B*}(y) = \sup_{x \in X}[\mu_{A*}(x) \star \mu_{A \to B}(x, y)] \tag{2.17}$$

2. A second approach was introduced by Mamdani in [19] whilst presenting a fuzzy controller for a steam engine and boiler. The difference to be observed in Mamdani's work is the use of the *min* operation as an implication operator, he obtains this by viewing implication as a binary fuzzy relation (such as fuzzy conjunction). He goes on to integrate this approach into generalised modus ponens with the argument: "Relations of higher order than two can be similarly defined. For example, "If A then (if B then C)" is given by the Cartesian product $A \times B \times C$. And now given A' and B' the value of C' is inferred to be"

   $$C' = \sum_k \max_{ij} \min u_i' u_i v_j' v_j w_k$$

   Subsequently Larsen [17] proposed that the product operation could also be used in place of the minimum operation. Mendel [2] states that both are frequently referred to as *Mamdani implications* and that this extends to any t-norm operation as shown in equation 2.18.

   $$\mu_{A \to B}(x, y) = \mu_A(x) \star \mu_B(y) \tag{2.18}$$

   During the introduction of the paper "Efficient Algorithms for Approximate Reasoning" [20] van den Broek refers to the min case of the Mamdani implication as the *interpolation method*.

The Mamdani implication is used to form a *fuzzy logic system*, identified by Mendel [2] to also be known as fuzzy-rule-based systems, fuzzy expert systems, fuzzy models, fuzzy systems or FL controllers. The established configuration for a FLS is shown in Figure 2.2.

It can be seen that this configuration describes a classical control system with a crisp input x and a crisp output y. So far in this literature review we have covered the necessary fuzzy techniques to support the implementation of a fuzzifier (the application of a domain value to a fuzzy set) and inference as just covered. Next the notion of rules are introduced as visualised by Bart Kosko and subsequently defuzzification and several approaches to performing it are surveyed.

The term rule is used to describe the pairing of an antecedent tautology of fuzzy sets paired with a consequent set. Through fuzzification of all input variables in the tautology and combination of their degrees of membership using t-norms and t-conorms we call the resultant degree of membership the firing strength of the antecedent. This corresponds to the degree of truth of the A* term in the generalised modus ponens rule. Implication is then performed as covered and we obtain a resultant consequent fuzzy set.

Bart Kosko discusses a 'patch' notion to describe the relationship caused when several rules have a non zero firing strength for any given set of inputs, this is a very likely scenario in a fuzzy system. To deal with it aggregation of the consequent fuzzy sets (once then have had implication operation applied to them with the firing strength for their corresponding antecedents) must be performed to obtain a final output fuzzy set for defuzzification. Aggregation of fuzzy sets is performed by obtaining the union of all sets with some t-conorm operator.

With the resultant aggregated set a defuzzification is the final stage in a standard fuzzy system, this returns a crisp value in the domain of the consequent sets. Many techniques are available for defuzzification and we survey two now;

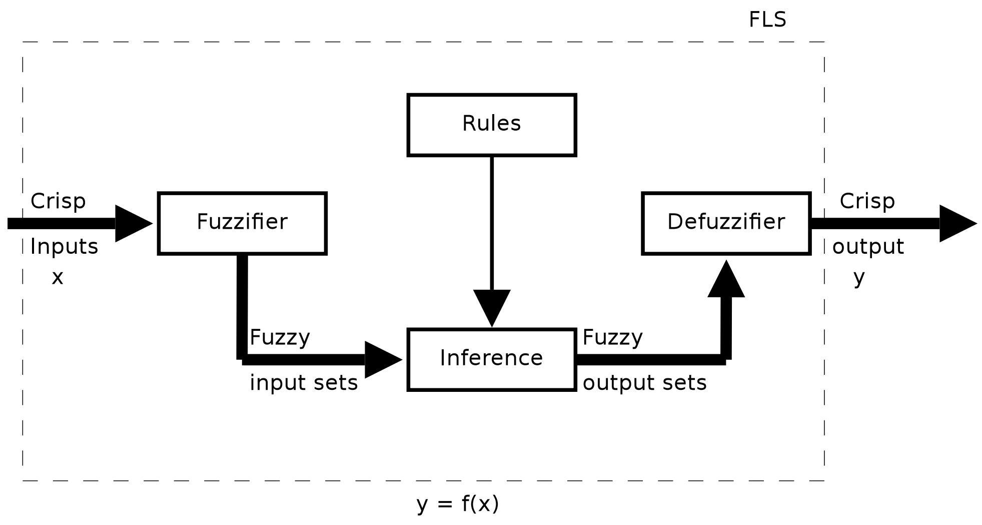

Figure 2.2: The components of a fuzzy inferencing system [2].

- Taking the centroid of a fuzzy set is a geometric analysis which obtains the centre of mass of the fuzzy set were it to be a physical plane. Inspecting the graph of an aggregated set of consequent fuzzy sets plotted with domain along the x axis and degree of membership on the y axis one can see this is a very intuitive approach.

- A second common class of defuzzification techniques involve the maxima of a fuzzy set, defined in [21] as equation 2.19.

$$\forall m \in \text{maxima}(A).\forall x \in X.\mu_A(m) \geqslant \mu_A(x) \tag{2.19}$$

  Because the maxima equation still returns a set of domain values (all those with the highest degree of membership in the set) we must select one specific value to return. For this there exist three common options:

  – Smallest of maxima (som),
  – Mean of maxima (mom),
  – Largest of maxima (lom),

Whilst implementation techniques are not to be mentioned in this section it is important to note here the two approaches when performing defuzzification, these are either the discretisation approach or numerical methods. In general we would say that the latter is preferential due to the efficiency of its implementation but it is not always possible. Critical to allowing it is whether the underlying geometric structure of the fuzzy set to be defuzzified has been maintained, for example this could comprise of :

- A number of x/y coordinates (where x is the domain value and y is the degree of membership) indicating segments which are linear in between.

- For a Gaussian function we may store the centre point, c, and the height.

If such a representation is available along with a known domain range then adopting a numerical method for solving a geometric equivalent of the defuzzification approach offers a computationally cheap way to obtain a defuzzified domain value. However if this is not the case the discretisation approach operates by obtaining a finite sampling of the domain at uniform intervals, the crisp value can then be calculated via summation of these domain value / degree of membership pairings as per the defuzzification technique. With the latter approach one is required to choose the number of domain samples they shall use; the choice is arbitrary offering a trade off between greater accuracy with a larger cardinality or less computation at lower speeds.

2.3 Functional approaches

Two publications on fuzzy logic which use functional programming languages shall be reviewed in this section. Firstly we cover a paper by P. M. van den Broek which present a implementation of fuzzy logic systems in the Miranda programming language. Alongside the implementation is discussion of how the computational complexity of intersection and implementation functions can be reduced by restricting membership functions to those which are piecewise linear and continuous.

Secondly a paper by Gary Meehan and Mike Joy titled "Animated fuzzy logic"[21] is reviewed. This paper adopts an aim very similar to the goal of this project, that is to "see how the high-level, declarative nature of a functional language allows us to implement easily and efficiently solutions to problems using fuzzy logic". This paper provides an implementation in Haskell alongside discussion of its functionality.

Using piecewise linear membership functions

Entitled "Efficient Algorithms for Approximate Reasoning"[22] P. M. van den Broek shows a novel approach to computing generalised modus ponens (in Section 2.2.2) where the membership functions used are *continuous piecewise linear functions*, for the remainder of this report we shall use the term PLF to denote such a function.

The paper asserts a function to be a PLF if "it is linear in all but a finite number of points". Consider a PLF μ_{PLF} comprised of three points, we shall denote the domain values with x_0, x_1, x_2 and the co-domain values [5] with y_0, y_1, y_2.

These points split the domain in to linear intervals and to calculate a value $\mu_{PLF}(x)$ for some $x_0 \leqslant x \leqslant x_2$ there exists two cases:

- If x is equal to some x_n then trivially we have equation 2.20

$$\mu_{PLF}(x) = y_n \tag{2.20}$$

- Other wise the value may be computed using the equation given in 2.21.

$$\mu(x) = y_0 + \frac{(x - x_0) \times (y_1 - y_0)}{(x_1 - x_0} \tag{2.21}$$

Formulas 2.20 and 2.21 are taken directly from van den Broek's implementation of the function *apply* which takes a list of tuples of the form $[(x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)]$ representing the PLF and a value for x.

While one is not provided in the paper it is felt that a graph aids the understanding of the relationship between the PLF, its list representation and the *apply* function. An example graph for such a graph is given in Figure 2.3.

By restricting the membership functions to only PLFs van den Broek shows how the computational complexity of the generalised modus ponens operation can be reduced. The paper notes that "the computation of a single value B'(y) involves functions over a continuous interval, and usually involves the application on numerical approximation techniques"[6] but then simply states that "the complexity of the computation can be reduced considerably", little discussion on why this is the case is given other beyond commenting that the reduction occurs from the fact that no discretization is necessary. After study of the implementations provided in the paper it becomes clear that the author's claims of reduced complexity lie in the use of *numerical methods* to perform the computation. Such an approach and how it differs from the sampling method were discussed in section 2.2.1. In the paper van den Broek states that "in the case where domains are finite intervals of real the real numbers [..] one cannot compute B' in general". This is to say when computing a generalised modus ponens we must discretise the fuzzy sets we are operating on (which in the paper are labelled A, A' and B) before computation can take place and that increasing number of *samples* we take during this process increases the computation time. By using a numerical approach only possible when PLFs are used the author eliminates the need to perform such discretisation when computing values for B' and thus reduces the computational complexity drastically.

[5] Which in the context of a fuzzy membership function indicate degrees of membership.

[6] Here B'(y) denotes the membership of y to the set B', equivalently $\mu_{B'}(y)$

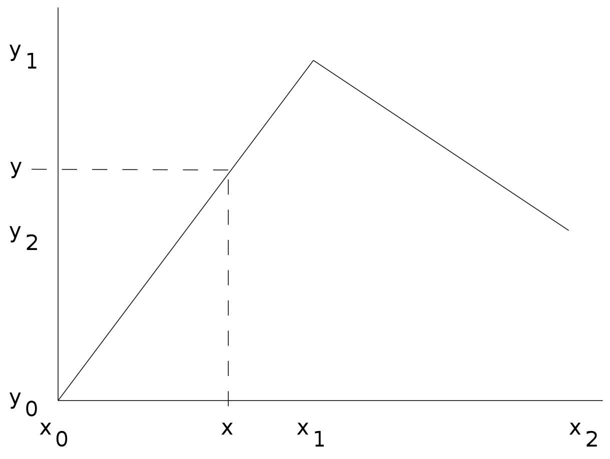

Figure 2.3: A piecewise linear function

A second desirable factor of van den Broek's approach is that one may approximate continuous input functions using PLFs. In the paper this fact is noted without any investigation, it follows however that one must decide a number of linear intervals to *sample* the input function over and that as this number increases so will the complexity of computing a generalised modus ponens using it.

In the paper the functional language Miranda is used with the justification that it "provides an excellent formalism for the notation of algorithms, and this notation is executable". Whilst this is true it is felt that there is another substancial advantage to using a function language which van den Broek does not mention, namely that of *partial function application.* This concept is covered in greater depth in Section 2.4.1 but consisely it is the notion that given a function which takes N arguments if one applies only one argument to it they obtain a function which takes $N-1$ (the remaining) arguments[7]. Because van den Broek's approach uses a list of 2-tuples to represent a PLF a function must be provided to turn a list into the membership function it describes (a function which takes a member of the universe and return its degree of membership). The function provided in the paper to perform this is called *apply* which takes the list describing the PLF and a domain value. However in a functional approach the application of *just* the the list to *apply* returns a one argument function which takes the domain value; this function then *is* the membership function.

A Haskell implementation of fuzzy logic

In "Animated fuzzy logic"[21] the reader is taken through the implementation, in Haskell, of of the components of a fuzzy logic system. This culminates in the presentation of three example systems [8] which utilise the implementation. The following areas are covered:

- Representation of a degree of membership using the Haskell *Double* type.
- Overloading of crisp (boolean) logic operators with ones which use a range of real numbers as the valuation set and so operate on fuzzy sets.
- How parametric definitions may be implemented.
- Calculation of the domain, support and fuzziness of a fuzzy subset.
- The use of higher order functions to provide functions over fuzzy sets (such as conjunction).
- Linguistic hedges and fuzzy numbers.
- Defuzzification.
- A Mamdani type fuzzy system including the representation of rules and an implication operation.

[7]Clearly in languages which are not functional this would result in an error.

[8]A shoe size fuzzy system, a controller for a shower and a goods pricing system.

The approach taken in reviewing this work is to present the Haskell code verbatim from the paper and alongside we shall review it. Thus it is implied for the remainder of this section that all Haskell code is taken directly from [21] and that conversely all commentary is written by the author unless explicitly stated otherwise. Lastly it should be noted that the difference in presentation style between the original paper and below is due to the use of a text preprocessor but that the source code from which it is generated is identical to that found in "Animated fuzzy logic".

In taking advantage of Haskell's ad-hoc polymorphism[23] (covered in Section 2.4.1) the existing logic functions which provide conjunction, disjunction, negation and operations over lists are lifted to a class definition such that they may be overloaded:

```
import Prelude hiding ((∧), (∨), ¬, and, or, any, all)
class Logic a where
    true, false :: a
    (∧), (∨) :: a → a → a
    ¬ :: a → a
```

We can see that this minimal definition allows the implementation of six additional logic operations. By specifying a *Logic* class in this way we need only provide the above definitions for a logic's valuation set and the six definitions below come for free. It is this kind of generalisability which shall support our argument that the functional approach to fuzzy logic is an advantageous one.

```
    and, or :: Logic a ⇒ [a] → a
    and = foldr (∧) true
    or = foldr (∨) false

    any, all :: Logic b ⇒ (a → b) → [a] → b
    any p = or ∘ map p
    all p = and ∘ map p
```

Now the authors demonstrate how the type *Double* may be used to create an **instance** of the *Logic* class we have defined. To do this they provide a complete *minimal definition* of the class, these read very naturally as shown below and from this declaration we can make the following observations:

- We are defining a logic where the valuation set is the set of real numbers. This notion was covered in Section 2.2.1.
- The declarations of *true* and *false* suggest the valuation is set to $x \in ,0 \leqslant x \leqslant 1$. However it is worth saying that this itself does not enforce that this will always be the case.
- We can see that the *min* and *max* functions are used. Overloaded for the type *Double* these take a pair of numbers and return the smallest and largest respectively. Thus we can see the author's have chosen to use Zadeh's original operations [9].
- Again for negation (supported by the ¬ function) Zadeh's one minus function is used.

```
instance Logic Double where
    true = 1
    false = 0
    (∧) = min
    (∨) = max
    ¬ x = 1 − x
```

Next a type synonym is introduced, we should read the line below as "something of type fuzzy a is a function to something of type a to a *Double*". Here a is a type variable and can be replaced with the any type, further this type denotes the domain of the function. In its context within the fuzzy system perhaps a more descriptive name for the type synonym would be *MembershipFunction* or *FuzzySet* such that it would read "a *FuzzySet* with a domain of type a".

```
type Fuzzy a = a → Double
```

It is also worth noting that this demonstrates the use of functions as first class variables as is covered in Section 2.4.1 it is sufficient here to say that it allows functions to be returned from functions and this is shown below.

```
up :: Double → Double → Fuzzy Double
up a b x
    | x < a = 0.0
    | x < b = (x − a) / (b − a)
    | otherwise = 1.0
```

Here we see that the *up* function takes two *Double* arguments and returns "a membership function who's domain is of type *Double*". This idea is presented elegantly on the next two lines where it is shown how a fuzzy set describing a *profitable percentage* could be shown. It is worth noting that in Haskell syntax we express the type as *profitable* :: *Percentage* which reads "profitable is of type percentage".

```
type Percentage = Fuzzy Double
profitable = up 0 15
```

Following some discussion on calculating the fuzziness of a fuzzy set higher order functions [9] are used once again in extending the operations defined previously to apply to fuzzy sets (in fact the definition given also generalises to crisp sets).

```
instance (Logic b) ⇒ Logic (a → b) where
    true = λx → true
    false = λx → false
    f ∧ g = λx → f x ∧ g x
    f ∨ g = λx → f x ∨ g x
    ¬ f = λx → ¬ (f x)
```

The coverage of fuzzy numbers and linguistic hedges which follows shall not be reviewed because it is not the intention of this project to use such techniques in establishing the case for using a functional approach to fuzzy logic.

The next area of interest to us is defuzzification for which four methods are provided in the paper:

- The centroid,
- Smallest of maxima (som),
- Mean of maxima (mom),
- Largest of maxima (lom),

Each of these four defuzzification techniques were surveyed in Section 2.2.3, now the authors of the paper show how they may be easily implemented in Haskell. Firstly the equation for the discrete case of the centroid function is stated, presented here in equation 2.22, the discrete case is used because representation chosen requires one to be specified in order to perform defuzzification. So for a fuzzy set with membership function *Fuzzy a* (that is a fuzzy set who's domain is of type *a*) we provide a domain as a list of items, all of type *a* ([*a*] in Haskell notation). A type synonym *Domain a* is defined to clarify this as shown below:

```
type Domain a = [a]
```

The implementation given is particularly pleasing in demonstrating the ease and symmetry with which mathematical equations can be expressed in the Haskell language. Thus we see the Σ_X replaced with the *sum* function and the fraction representation translated to the infix division operation /. Further given the domain X as the argument *dom* we may generate the co-domain (that is $\mu_A(x), \forall x \in X$) by mapping the membership function f over the domain. With the result bound to *fdom* the product $(x\mu_A(x))$ is obtained by zipping the lists together with the product function $(*)$, the zipping operation applies this product function to each pairing of domain and co-domain value.

[9] Defined by Thompson in [24] thus: "A function is higher-order if either one of its arguments or its result, or both, are functions.

$$\frac{\Sigma_x x \mu_A(x)}{\Sigma_x \mu_A(x)} \tag{2.22}$$

```
centroid :: Domain Double → Fuzzy Double → Double
centroid dom f = (sum (zipWith (∗) dom fdom)) / (sum fdom)
    where fdom = map f dom
```

At this point it is worth noting one disadvantage of choosing the fuzzy set representation used (that is, one defined by a membership function of **type** *Fuzzy a* = *a* → *Double*). Namely that because we have no knowledge of the underlying nature of the function we cannot use numerical methods to compute its centroid. Due to this discretisation must take place and the accuracy of it depends on the on the cardinality of the domain set provided as covered in Sections 2.2.1 and 2.2.3.

Next the smallest, mean and largest maxima defuzzification operations are implemented. To support this a function *maxima* is defined and in doing so the author's introduce the *Ord* type class [10]. Inspection of the function's definition reveals that this is not actually required for this function to be correct, it appears to be present because the defuzzification functions which use it (*minmax*, *maxmax* and *medmax*) do require the domain to have an ordering. The function will be investigated now to show a demonstration of the ease with which functions can be generalised using polymorphism; a feature which is used in our argument for a functional approach to fuzzy logic.

```
maxima :: Ord a ⇒ Domain a → Fuzzy a → [a]
maxima dom f = maxima′ dom [ ]
    where : q
        maxima′ [ ] ms = ms
        maxima′ (x : xs) [ ] = maxima′ xs [x]
        maxima′ (x : xs) (m : ms)
            | f x > f m = maxima′ xs [x]
            | f x ≡ f m = maxima′ xs (x : m : ms)
            | otherwise = maxima′ xs (m : ms)
```

Note that the type of the *centroid* function was *Domain Double* → *Fuzzy Double* → *Double*, however for *maxima* the type given is *Ord a* ⇒ *Domain a* → *Fuzzy a* → [*a*]. This states that it may be a domain of any type provided that it is an instance of the *Ord* class (and also that the *Fuzzy* membership function is defined over the same type). Lets look at both cases:

- For the *centroid* function we must use the product operation (∗) on both the domain and the co-domain (a value from the valuation set). Because the latter is already of type *Double* from the application of a function of type *a* → *Double* to the domain we see that the second argument to (∗) is of type double. This forces the product functions first argument to also be a *Double* through the rules of type inference. This is covered in greater depth in Section 2.4.1.

- In the latter case we have the function *maxima* with type *Ord a* ⇒ *Domain a* → *Fuzzy a* → [*a*], firstly we survey its implementation and then discuss the erroneous *Ord a* class constraint.

  A suitable starting point is to ask why the type signature contains the type variable *a* and not *Double*, the answer being that unlike *centroid* at no point in the function definition is a domain type *a* used without being first applied to the membership function *f*. In following the list of domain values *dom* we see they are first passed to the auxiliary function *maxima′*, here they are de-constructed using the (*x* : *xs*) notation (such that *x* contains the first element and *xs* is the rest of the element). The function then maintains a list of the domain values which yield maxima in the (*m* : *ms*) list. Crucially however we see that all occurrences of *x* and *m* where they are supplied as arguments to *f*, in *f x* and *f m* respectively. We can infer then that the type *a* of the domain values (the (*x* : *xs*) and (*m* : *ms*) must be the same as the type *a* in the type of the membership function, namely *a* → *Double*.

With the *maxima* function now in place we now see the implementations for *minmax* and *maxmax*.

[10]If a data type is an instance of *Ord* then it has a natural total ordering and provides functions such as less than and greater than.

```
minmax, medmax, maxmax :: Ord a ⇒ Domain a → Fuzzy a → a
minmax dom f = minimum (maxima dom f)
maxmax dom f = maximum (maxima dom f)
```

Lastly the *medmax* function for which the author's have chosen to provide their own quick sort implementation. The reason for choosing quicksort demonstrates another desirable property of Haskell, namely lazy evaluation. Given the first three lines of the function definition:

```
medmax dom f = median (maxima dom f)
    where
        median ms = head (drop (length ms ‘div‘ 2) (qsort ms))
```

We see the *head drop* statement in place to skip the first half of the list and then take the first element (*head*) of the remaining half, thus giving us the median value of a sorted list. We know then that we only require the list to be sorted such that all values to the left of the middle element are less than it and all values to the right are greater. By combining the properties of the quicksort algorithm with lazy evaluation we can perform the minimal amount of sorting possible to satisfy this requirement; first observe the *qsort* function:

```
qsort [] = []
qsort (x : xs) = qsort [y | y ← xs, y ⩽ x] ++ [x] ++ qsort [y | y ← xs, y > x]
```

By evaluating the left most recursive call to *qsort* first (as is standard behaviour) we now show how computation halts as soon as the median element (the *head* of the remaining list once half has been dropped) is computed. In the example computation shown below we see the list $[6, 5, 2, 3, 9, 8, 7]$ being sorted, however because only the median (fourth) element is required (the number 6 in this example) the evaluation of *qsort* $[9, 8, 7]$ never needs to be performed. Note that the $y \leqslant x$ test in the function definition has guaranteed that $[9, 8, 7]$ are all greater than 6 as required.

				qsort [6, 5, 2, 3, 9, 8, 7]				
			qsort [5, 2, 3]				++[6]++	*qsort* [9, 8, 7]
		qsort [2, 3]			++[5]++	*qsort* []		
qsort []	++[2]++		*qsort* [3]					
[]								
		qsort []	++ [3] ++	*qsort* []				
		[]						
				[]				
						[]		
	2		3		5		6	N/A

The next section on the paper investigates fuzzy database queries and demonstrates how the system works and the use of linguistic hedges. As these are of no concern to the project we now address the following section on fuzzy systems. Here a Mamdani type fuzzy inferencing system is implemented (as covered in Section 2.2.3) by building on the previously defined *Logic* class with an implication operator. The operation is defined by the infix function ⇒ and from its type definition we can see it takes a *Double* (indicating the degree of truth of the antecedent [11]), something of type *Logic a* representing the consequent and returns a *Logic a* which is the result of the rule firing.

```
infix 0 ⇒
class Logic a where
    (⇒) :: Double → a → a
```

By specifying a low precedence infix operator in this way the authors provide a very readable implementation of the implication operation; one which may not be so attainable in a non-functional language.

[11]This is phrased in the paper as "The degree to which the action fires, the action being the assignment of a variable to a fuzzy subset".

The given *shoe_size* function is a fuzzy inferencing system, fulfilling all the requirement laid out in Section 2.2.3, namely:

- A fuzzifier provided by *down*, *tri* and *up* functions (all which take their parameters and return a function of type *Fuzzy Double*.
- A set of rules, which we shall cover below.
- Inference, provided by the implication operator ⇒ as we have just covered.
- Defuzzification, provided by the aggregation function (+) and the *centroid* function, the former is discussed now and the latter was covered previously.

```
shoe_size :: Height → ShoeSize
shoe_size h = centroid sizes (
    rulebase (+) [
        short h ⇒ small,
        medium h ⇒ average,
        tall h ⇒ big,
        very_tall h ⇒ very_big])
```

Looking at this definition gives justification that the implementation is readable as stated before. Specific arguments for this case are given now:

- In the antecedent components of the rules we can see the application of the crisp input *h* to the fuzzification (membership) functions *short*, *medium*,. . . .
- We can clearly see how the implication operator ⇒ takes place of the linguistic term "then" and the "if" part is implicit at the start of each rule in the list.
- The rule aggregation operation (+) is provided as the first argument to the *rulebase* function, making clear the function of this operator.
- As discussed previously given continuous fuzzy sets it is necessary to provide a domain to discretise over when performing defuzzification. We can see the *centroid* function is provided both a domain (the universal set of shoe sizes) and the aggregated output fuzzy sets from inferencing.

The author's of the paper go on to present to further examples of problems solved using the fuzzy logic system presented in the paper but review of these serves no benefit to the project and as such the review of the author's Haskell code shall finish here. In conclusion the facts most significant to the project are presented now:

- A new type class *Logic a* was introduced to allow the implementation of a multivariant (fuzzy) logic along side the existing boolean (crisp) one present in Haskell.
- Membership functions were modelled using functions of the type $a \to Double$ such that the domain could be of any type a.
- For a domain of type *Double* three types of parametric membership functions are provided: *up*, *down* and *tri*.
- Four defuzzification method are provided: The *centroid* function, it uses a discretisation approach and requires a sampling domain to be provided. We also see the three median functions *minmax*, *medmax* and *maxmax* supported by the *median* function.
- Aggregation of consequent fuzzy sets is provided by an unbounded addition operator, whilst this yields a resultant set which is not strictly a fuzzy set the author argue the benefit that "unlike union, when combining many sets the membership function of the result doesn't approach the constant function 1".
- The product operator is used as the t-norm in a Mamdani type implication provided by the ⇒ function.

- Sets of rules are stored as lists of applications of this implication function. The arguments are an antecedent consisting of the application of an input variable to a parametric membership function (such as *top*) and a consequent consisting of a parametric membership function.
- A function performing the task of a fuzzy inferencing system is constructed by applying a defuzzification function *centroid* to an aggregation of all the rules (performed using *foldr*).

2.4 The functional programming paradigm

2.4.1 Properties of functional programming

Firstly the author presents a review of the properties which characterise a typical functional programming language. These shall provide a basis for the justification of investigating functional approaches to fuzzy logic which is provided in Section 3.

The list of properties and the order in which they are covered is taken from chapter 2 of "Type Theory and Functional Programming"[25] by Simon Thompson, wherein each property is given a paragraph describing what functionality it provides. Here they shall be used here as a springboard from which to present a deeper review of each property including the advantages and disadvantages it brings compared again other programming paradigms.

First-class functions

A central theme to functional programming languages is the notion that functions themselves may be passed as arguments to other functions and returned from functions, this property is known as first order functions. If any of a functions arguments or its return type or both are a function we call a function higher order. The distinction to be made between languages in which functions are first order and those in which they aren't is phrased concisely in an article summarising the concept entitled "Persistent First Class Procedures are Enough"[26][12]:

> The procedures of Algol 60[19] and Pascal[26] can only be declared, passed as parameters or executed. However, as has been pointed out by Morris [16] and Zilles [29], to exploit the device to its full potential it is necessary to promote procedures to be full first class data objects. That is, procedures should be allowed the same civil rights as any other data object in the language such as being assignable, the result of expressions or other procedures, elements of structures or vectors etc.

Atkinson later states "Another advantage of having procedures as first class data objects is the possibility of having partially applied functions" and we shall review the concept of *partial application* now.

Whilst the notation is specific to the Haskell programming language the author feels a good intuition for the behaviour of partial application is presented in Simon Thompson's "Haskell: The Craft of Functional Programming" and the following is paraphrased from Section 10.4:

- We may denote a function's type using the syntax $a \rightarrow b$ this states the function takes one argument of some type a and returns something of type b.
- This can be extended to denote the type of a function which takes two arguments of types a and b and returns a c, thus: $a \rightarrow b \rightarrow c$.
- By observing that the function space symbol '$\rightarrow$' is *right associative* we see the previous type can be written $a \rightarrow (b \rightarrow c)$.
- So by supplying a function of type $a \rightarrow (b \rightarrow c)$ with an argument of type a, we see $(b \rightarrow c)$ is returned. This is a function from b to c and is equivalent to the original function *partially applied* with its first argument.

[12]In the paper the terms procedure and function are used interchangeably

Type systems

As alluded to in the previous section a key element in certain functional programming languages is a strong typing system. This enforces the requirement that a function's arguments must be of a known type and that attempting to evaluate function with a wrongly typed input yields an error.

The type of a function is referred to as its *type signature*, given using the $\rightarrow$ notation introduced in the previous section. Type signatures shall be used heavily in this report.

Whilst functional programming languages provide their own common types such as strings and floating point values more complex ones can be constructed and defined using algebraic data types. Typically a *type constructor* is used to allow the construction and deconstruction of these new types.

Even with algebraic data types the system is still very restrictive, Thompson[25] introduces the need for type polymorphism in the following way: "A potential objection to strong typing runs thus: in an untyped language we can re-use the same code for the identity function over type;[..]". The argument made here is that without out some form of extension a strong type system which demands all functions to have a given type (for example one which takes an integer and returns an integer) will lead to vast repeating of code, specifically in the case where the type of the argument is irrelevant (such as with the identity function).

A solution then is to allow polymorphic types such as $a \rightarrow a$ [13] in which a may be substituted with any real type (such as an Integer). "By allowing types and programs to be parametrized [in this way] by types, or even type constructors, it becomes possible to create highly reusable libraries, while maintaining the benefits of compile-time type checking" [28].

Two distinct varieties of polymorphism can be found, both described by Wadler and Blott[23] using the following examples:

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type.A typical example is overloaded multiplication of integer (as in 3×3) and multiplication of floating point values (as in 3.14×3.14).

Parametric polymorphism occurs when a function is defined over a range of types, acting in the same way for each type. A typical example is the *length* function,which acts in the same way on a list of integers and a list of floating point numbers.

When combining type variables as just covered with the property of partial application discussed previously one can demonstrate type inferencing, this is shown now by three examples:

1. Firstly we consider a replication function, its job is to take something (of any type) and an integer and return a list of the first argument repeated the second argument number of times. From this we can put forward the type signature $a \rightarrow \text{Int} \rightarrow [a]$ [14] , here we see one type variable a. Suppose we now partially apply this function by giving a first argument of type Char, as soon as this is done we can infer the type of what the function returns to be Int $\rightarrow$ [Char] because the a type variable has now become fixed to Char. This process is shown step by step now, we see the evaluation on the left and its type on the right:

$$\begin{array}{rl} \text{replicate} & a \rightarrow Int \rightarrow [a] \\ \text{replicate 'x'} & \text{Int} \rightarrow [\text{Char}] \\ \text{replicate 'x' 5} & [\text{Char}] \end{array}$$

2. Secondly we shall show a more complex type inference process involving two functions and two type variables such that the function returned after partial application still contains one type variable. To do this we shall take a function map which is common in functional programming; it allows a function from one type to another to be provided and then maps this function over each element in a list to return a list of identical length. We can give the type signature of map as $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ [15].

[13] In the context of the identity function this can be identified as the Hindley-Milner type system[27] in which the type is denoted $* \rightarrow *$.

[14] The square bracket notation indicates a list of multiple objects of one type, e.g. [Int] indicating a list of integers.

[15] Note the use of parenthesis around $a \rightarrow b$, so the first argument is a function from a to b. Were they not present it would indicate two arguments, the first of type a and second of type b.

We must now select a function to pass as the first argument to map (recall that by the property of higher order functions we may do this and the type of the first argument of map also demands a function from a to b). For this we choose a simple length function, its type is $[c] \rightarrow \text{Int}$ indicating it takes a list of any type and returns an integer which is the length of the list given as the first argument.

Looking at the partial application evaluation step by step, we see how type inference is performed substituting type variable a for the list of any type $[c]$ and type variable b for the real type Int:

$$\begin{array}{rl} \text{map} & (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{length} & [c] \rightarrow \text{Int} \\ \text{map length} & [[c]] \rightarrow [\text{Int}] \end{array}$$

At this stage we have partially applied map to length, we can see the returned function takes a list of lists (of any type c) and returns a list of integers. Taking the application further we can consider type variable c to be a character, thus a list of characters $[c]$ is a word, making the type $[[c]]$ to be a list of words. Now we clearly see how the type of the returned function indicates the semantic nature of the function, it counts the number of letters in each word in a list:

$$\begin{array}{rl} \text{map length [['g','o'], ['b','a','r']]} & [\text{Int}] \\ [2, 3] & [\text{Int}] \end{array}$$

In this section we have introduced and demonstrated the power of a strict type system combined with polymorphism, in demonstrating how type inferencing can add valuable semantic value to applications of functions, specifically higher order functions. We return to this as part of our argument for the use of the functional paradigm later in Section 3.

Type classes

Building on the properties of polymorphism introduced in the previous section we now look at a concept which shares some ideas with the object orientation concept of interfaces. In the functional programming paradigm we may provide a type class which specifies a list of functions an instance of it should support, much in the same way as an interface outlines the methods an implementing class must provide in the object oriented paradigm.

Because in functional programming type variables provide the functionality of generics it is not surprising they are used in defining a type class, this is best shown by an example. Consider a class which defines equality, we might then state that for some type a to be an instance (that is provide the functions) of the equality class it must provide a function $==$. However this is little use without knowing in advance what the type of the $==$ function is, to solve this we make the type signature of the function an explicit part of the class definition. Here we use type variables and obtain a type signature $a \rightarrow a \rightarrow \text{Bool}$, so the function takes two arguments of the same type a and returns a boolean value (indicating whether they are equal).

Any type now wishing to become an instance of the equality class must provide an implementation of the $==$ function which has the type $a \rightarrow a \rightarrow \text{Bool}$. The advantage of this becomes clear when one considers a function which tests whether a list contains an argument. We might suppose its type is $a \rightarrow [a] \rightarrow Bool$, however how are we to know whether each element of the list is equal to the first element, clearly we need a function to test equality such as our $==$ function. This is guaranteed by any type which is instance of the equality class. We now put an extra qualifier on the type $a \rightarrow [a] \rightarrow Bool$ to state that a must be a instance of the equality class.

This concept can be taken one step further by providing default function definitions in the class itself. We can do this to define a not equal operator in terms of our equality function $==$ by stating that if it returns true then we should return false and vice versa. Crucially we see that this does not require each instance of the equality class to provide there own not equal function because we may derive it from $==$.

Algebraic types

Another property of many functional programming language type systems is allows the one to create their own type in terms of a composition of existing types, however for the purpose of this report we shall only consider the special case where only one type is used. To facilitate this one needs some form of type constructor which indicates the type of what it returns, consider a new algebraic type for a person; we may derive a type constructor for this called Person.This constructor also has a type, a natural one may be String → Person, so we see we provide the type constructor with an argument of string type (a name) and it returns something of type Person. Language specific notation then allows one to obtain the string component back through a technique such as pattern matching, for example (Person s) where we now define s to be the name.

Algebraic types become useful when we wish to utilise polymorphism but require different implementations for one basic type such as a string. Now we can wrap strings up as different types allowing them to behave differently but still having access to the underlying string type when required.

2.4.2 Haskell

Syntax and notation

For the benefit of the reader Haskell specific syntax and notation which shall be used in this report is covered briefly now.

Function composition is denoted in Haskell with $\circ$, thus the statement $f\ (g\ x)$ can be written as $(f \circ g)\ x$. It is common to define functions in terms of other functions, thus if one commonly applies g to something and then applies f to the result they may define a new function $h = f \circ g$ such that now $f\ (g\ x) = h\ x$. Observe that in the definition of h the argument x was not mentioned, this is known as *pointfree* style.

Dollar notation is used in Haskell to force the changing of application priority, usually to reduce the number of parenthesis required. Because function application has the highest priority an expression such as $f\ g\ h\ x$ is evaluated as $((f\ g)\ h)\ x$ but often we require an order such as $(f\ (g\ (h\ x)))$ and to achieve this the dollar is used thus: $f\ \$\ g\ \$\ h\ x$. Note that this could also be expressed in pointfree style.

List comprehensions are a syntactic feature in Haskell which are used to generate lists, the notation is very similar to the set notation given in 2.2.1 They shall be used extensively throughout this report and as such familiarity with them is important. The comprehension appears between two square brackets is is composed of a statement of what is to be returned, generators and filters. The returned component occurs first followed by a vertical bar. After the bar a list of generators and filters are given, separated by commas. Two examples are given below to demonstrate the notation:

- Given a list of the natural numbers between one and ten called xs we can return all the ones greater than five, multiplied by two using this comprehension:

  $$[2 * x \mid x \leftarrow xs, x > 5]$$

- A more complex comprehension used later in the report generates a list from a list of tuples called *plf*. We can see each element in *plf* is extracted in turn such that its two component values are available as x and y. The former is returned only if it corresponding y values is greater than a value *maxy*.

  $$[x \mid (x, y) \leftarrow plf, y \geqslant maxy]$$

List enumerations are a concise way to specify an arithmetic sequence as a list provided that the type of the item in the list is enumerable. To specify a range with a beginning and end of 4 and 7 the notation $[4 \mathinner{..} 7]$ is used and accordingly that expression evaluates to the list $[4, 5, 6, 7]$. Additionally the 'step size' can be specified using a comma thus: $[4, 4.5 \mathinner{..} 6]$ which evaluates to the list $[4, 4.5, 5, 5.5, 6]$.

Modules in Haskell allow separation of code and hiding in a similar fashion to most languages. A module may be imported with the key word **import**, this bring all the functions exported by the following module names in to scope. In certain cases this may cause a name conflict in which case the statement **import** *qualified*, this requires all functions in this module to be referenced as *ModuleName.FunctionName*.

Rather than importing all the functions from a module one may specify a list of ones to import (provided they are exported by the module) using this notation:

import *ModuleName* (*import1*, *import2*)

Alternatively one may import all the exported functions of a module *except* specified ones using the *hiding* keyword before the list.

To define a module the **module** *MyModule* **where** structure is used to prefix all the functions which belong to module *MyModule*. To control which functions are exported by a module (that is those which are imported into scope in another module which imports the one in question) a list may be provided in parenthesis following the module declaration thus:

module *MyModule* (*export1*, *export2*) **where**

If a list is not provided then all the functions in the module are exported.

Type synonyms and new types are two features of Haskell, the former of which exists only to improve the readability of code and the latter of which supports the creation of entirely new types, defined in terms of existing types.

The former, called type synonyms, allows one for example to generate a new type, for example *Name* which is identical in all aspects to the standard Haskell type *String*. Indeed the Haskell compiler will actually convert all occurrences of *Name* back in to *String* during compilation. As will be seen however this feature does improve the readability of type signatures by making the semantic nature of the arguments of a function more verbose.

In certain cases one may wish to define an entirely new type in terms of an existing type, as identified in Section 2.4.1 we should use algebraic data types. An example may be a new type person, this new type consists of a name given by a *String* and an age given by an *Int*, we would define this new type thus: **data** *Person* = *Person String Int*. With this definition in scope we now call *Person* the type constructor, and by evaluating an expression such as *Person*"*Dave*" 23 a *Person* is returned. An example usage of this type could be a function *birthday* which adds one to a persons age. We may define it thus:

birthday :: *Person* → *Person*
birthday (*Person n a*) = *Person n* (*a* + 1)

Firstly we see the type signature stating that the function takes a *Person* and returns a *Person*. On the following line we see how the types (*String* and *Int*) which compose the *Person* type are exposed by pattern matching the *Person* type constructor. To the right of the equals sign we see the type constructor used again to wrap the original name and new age back up to a *Person* type.

Type classes in Haskell are implemented with two components, a **class** definition and a number of **instance** implementations for different types which provide the functions for this class. The theory behind type classes and an example is given in Section 2.4.1.

If we consider a class seniority which states that any type which is an instance of it provides a function to see if the first argument is considered senior to the second. Firstly we open the class definition with the line:

class *Senior a* **where**

An important component in this line is the type variable *a*, in the remainder of the class definition any reference to the type *a* will mean the specific type whom is an instance of this class; this shall become clear when we see a **instance** of this class shortly.

Because this is a class definition we only need to [16] specify the type signature for the function

[16]Except in the case where as default function is used as covered in Section 2.4.1.

thus:

$$senior :: a \rightarrow a \rightarrow Bool$$

This function takes two arguments of type a and returns a boolean value. Importantly the a type here is equivalent to to the type of the instance as mentioned before. So were we to make our *Person* type from the previous section a instance of the *Senior* class with the statement:

instance *Senior Person* **where**

We can see that quite naturally the type variable a in the class definition is now the actual type *Person*, as such the a in the type signature for *senior* also becomes *Person*. So we now know that we have to define a function called *senior* with type $Person \rightarrow Person \rightarrow Bool$.

Compilers

During development of this project the author shall be using a Haskell compiler and an interpreter. We briefly introduce them now.

The Glasgow Haskell Compiler [29] or GHC is a described by its website as "a state-of-the-art, open source, compiler and interactive environment for the functional language Haskell". It provides many features which are not part of the standard Haskell 98 and some of which are used during this report. All provided definitions in this report may be compiled with GHC provided the switch `-fglasgow-exts` is used.

Hugs [30] is an interpreter for Haskell which allows modules to be loaded and functions to be executed manually, this allows for a fast development cycle by allowing components to be tested individually. Below we see a sample Hugs session where the *List* module is loaded and the function *intersect* is used:

```
Hugs.Base> :load List
List> intersect [3,1,4] [1,5,9]
[1] :: [Integer]
```

Note that one feature of Hugs (which is disabled by default) is that it reports the type signature of the result of the computation following the result. When partial application is performed we can see this being of use because the actual result of a partial application cannot be shown. Taking an example from Section 2.4.1 we can see the type inferencing discussed performed by Hugs as it reports:

```
Hugs.Base> map length
ERROR - Cannot find "show" function for:
*** Expression : map length
*** Of type    : [[a]] -> [Int]
```

One further advantage of Hugs is its ability to detail the number of reductions performed and cells used during an evaluation. The technical nature of these counts is not covered within this report but later we shall use them to obtain a rough guide to the computational complexity of different function evaluations by quoting the number of reductions reported by Hugs.

Chapter 3

Discussion

3.1 Summary of chapter

In the first section the aims and objectives of the project will be discussed in more detail. Central to this discussion will be the forming of an argument for the use of functional programming languages in fuzzy logic systems. To achieve this the author utilises the knowledge ascertained during the literature review conducted in Chapter 2 to address the motivations presented at the outset of the project in Chapter 1.

With a more rigorous and considered analysis of the project motivations in place the author shall establish what exactly is to be shown by the report. It is the purpose of this section to introduce the reader in to functional programming approaches to fuzzy logic. Firstly by detailing supporting arguments for the claims made in the first section (namely the advantages of the functional paradigm) and secondly by discussing how previous works (namely those reviewed in Section 2.3) may be built upon.

The conclusion of this chapter focusses on what demonstrable goals are to be realised in the implementation stage of the project. Here the author concentrates on what specific properties are desirable in the system the project intends to produce. Furthermore the aspects discussed in this section should provide an indication of the degree of success of the implementation component, as such they shall be revisited in evaluating the work. It is not intended here to provide a technical overview of the goals, this shall be introduced as part of a methodology for implementation in the next chapter. Instead this discussion shall reflect on the points raised during Sections 3.2 and 3.3.

3.2 Review of objectives

At the outset of the report in Section 1.1 we observed several reasons for investigating fuzzy logic systems in a functional programming language. One was the lack of recent research in to the field available despite the increasing popularity of the paradigm, specifically from the Haskell community. It is reasonable to assume that this popularity stems from attractive features of the paradigm and so by exploiting them in a fuzzy system we shall gain insight in to what makes them attractive.

During the literature review in Section 2.3 two papers were covered detailing two different approaches to implementing fuzzy logic in functional languages. In the paper "Animated fuzzy logic" the functional techniques of lazy evaluation and ad-hoc polymorphism were used. The former can be seen to yield an efficiency advantage whilst the latter is shown through the use of type classes in the *Logic* class. The second paper by P. M. van den Broek covering piecewise linear membership functions demonstrate two other benefits of the functional paradigm. Firstly the syntax as described by by the statement "an excellent formalism for the notation of algorithms, and this notation is executable"[22] and secondly the use of partial application in the use of the *apply* function.

Given the motivations specified for this project it seems appropriate to build on the successes of these two papers, specifically those which are a result of properties of the functional paradigm. Because we intend to develop one set of modules for the Haskell language the work shall be primarily be to bring together both approaches. For this objective to be achieved successfully we would expect the produced system to allow further representations to be added to the system by abstracting the underlying fuzzy set representation out.

Making this abstraction work well and to the advantage of a user of the system shall provide additional scope for the use of properties specific to the functional paradigm. Now we go on to discuss what can be shown by this report to demonstrate that a fuzzy system which supports multiple fuzzy set representations is well suited to the functional paradigm.

3.3 What will be shown

As was made apparent in the previous section there are two distinct areas of concentration for the work of this report. Firstly we shall investigate further the potential for extending the work of Meehan and Joy and van den Broek. Secondly it is our intention to investigate what specific properties of the functional paradigm are beneficial and potentially damaging to the objectives of our implementation. We discuss each of these two areas now:

3.3.1 Development of a fuzzy system

It is proposed that to achieve the goals of building on the two reviewed papers and demonstrating the advantages of the functional programming paradigm we shall develop a single system which unifies the approaches taken in both papers. We shall state that unification of the two approaches has been achieved when a user of the system [1] may use both of them interchangeably without altering large volumes of the code which implements the system. How we define a large volume shall be best expressed in terms of comparison with achieving similar results in other paradigms; thus we cover this in our evaluation in the final chapter. It is assumed that fulfilling this requires that the developed fuzzy inferencing system shall provide an abstraction between the system itself and the underlying fuzzy set representations.

The advantage of such a system would be the algorithmic performance gains which may be found by selecting a fuzzy set representation appropriate to the problem domain. It is to be shown that several features and properties of the functional paradigm make the use of a fuzzy system with interchangeable representations more manageable than would be possible in other paradigms; we survey the these features and properties fully next in Section 3.4.

A further development put forwards is the support for the use of multiple representations within one fuzzy system. By doing this we allow the best possible trade off between the performance gains discussed and the suitability of a representation to the problem. To clarify how this is possible Figure 3.1 is provided in which we see a diagram of the components of a Mamdani fuzzy system as covered in Section 2.2.3 with the addition of two bounded areas indicated by α and β. Both these areas contain the fuzzy sets and inside both these sets are manipulated, however at the location where α and β join (during the implication from antecedent sets to consequent ones) only the firing strength of the rules is transferred. Because this strength is only a real number and not a fuzzy set we may utilise a different representation on either side.

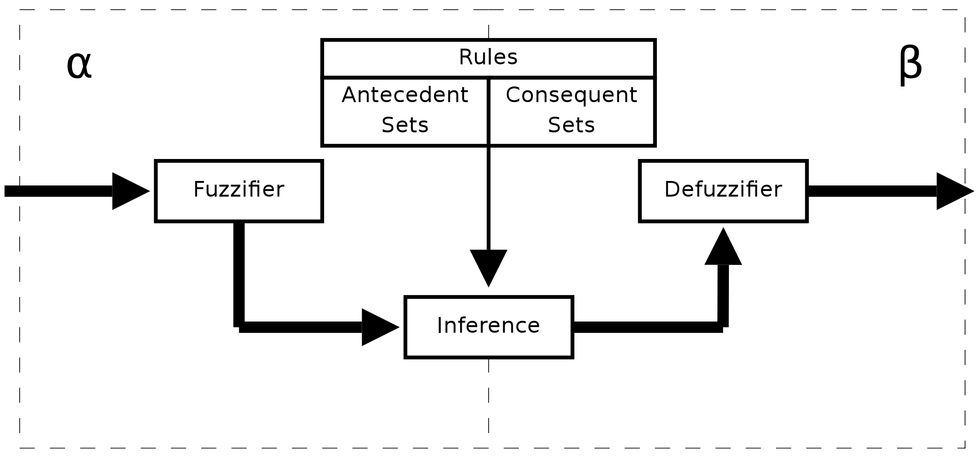

Figure 3.1: The use of two alternate representations (α and β) in one Mamdani system.

3.3.2 Use of the functional paradigm

Here we look at the properties of the functional paradigm reviewed in Section 2.4.1 and present some concrete examples of where they might be useful in a system such as the one discussed in the previous section.

- One difference observed between Meehan and Joy's approach and the one taken by van den Broek is that in the former the representation *is* the membership function. By contrast in the latter only a representation of a fuzzy set was given; we could say that in this way we have lost the execute-ability of the representation. The functional paradigm gives us an elegant solution to this as demonstrated by van den Broek with his *apply* function. By only partially applying it with the tuple list representing the fuzzy set we obtain a function still expecting the domain

[1]Who is assumed to be familiar with Haskell.

value, this is a membership function which we can use as a component in other functions in our fuzzy system.

- Another area in which the functional paradigm gains merit is allowing abstractions of patterns to be defined easily. One area in which we intend to exploit this is separating co-domain operators (such as *maximum*, *minimum* and *product*) from the underlying representation used by the fuzzy set they are operating on. A system which is able to do this will be greatly simplified in implementation by using type variables and type classes to specify exactly which functions are to be used when this abstraction is applied to a specific representation.

- Due to the declarative style possible in the functional paradigm it is often noted how close function definitions appear to there equivalent expressed mathematically. To cite an example here we consider the standard formula given for the defuzzification by centroid of a discrete fuzzy set thus:

$$\frac{\sum_{i=0}^{N} x_i \times \mu_A(x_i)}{\sum_{i=0}^{N} \mu_A(x_i)}$$

  To demonstrate how this may be defined in a functional language we consider the following Haskell definition, we suppose that a list of domain values are available as a list *xs* and their corresponding degrees of membership in a list *xas*.

  sum (*zipWith* (∗) *xs xas*) / *sum xas*

  Looking at this function definition we can easily see the *sum* function taking the place of $\sum$ and the infix division function / providing the fraction. We use the *zipWith* function to pair up all the elements of the two lists *xs* and *xas* (that is every domain value and its degree of membership) and use the given product operator (∗) to combine them as given by $x_i \times \mu_A(x_i)$.

3.4 Demonstrable work

Having discussed and explored the potential for further development of a fuzzy system in the Haskell language by extending previous work we now describe with a finer granularity exactly what is to be done.

Critical to the success of the project is its ability to convince the reader that features specific to the functional paradigm have made both the implementation and subsequent use of the system easier than in another paradigms. We have reviewed some of these features and how they apply to a fuzzy logic system in Section 3.3.2 and thus we shall put the emphasis on these when we demonstrate how a fuzzy system of the kind discussed in Section 3.3.1 may be implemented. Now we discuss exactly what properties we shall exploit and where:

Type classes will allow us to rigorously define what functions a fuzzy set representation must provide. By doing this we offer a convenient interface for people to develop the work further in the future.

Partial polymorphism will be used to allow the overloading of functions such that they can operate on any representation. Again this removes future reimplementation of functions specific to a fuzzy system and not to the underlying representations.

Higher order functions and partial application will be used to demonstrate how through application of a function similar to van den Broek's *apply*, a representation of a fuzzy set can be affectively turned into a membership function. This would simply not be possible in a paradigm without higher order functions, where the returning of a function from a function is not possible.

Mathematical notation is used heavily in fuzzy logic as is apparent if one glances over the review of the area presented in 2.2. We have already seen an example of who close a functional definition can be to its notation in a mathematical style.

Syntactic shorthand is used extensively in the Haskell programming language an example being the list comprehension notation. By making concise definitions using these we shall show how the code is more readable and thus more maintainable than possible in other paradigms.

In concluding this discussion section we outline the goals posed for the implementation, before pursuing these a more rigorous definition is given in the methodology chapter. Regarding fuzzy logic we intend to:

1. Bring together the work of Meehan and Joy's Animated fuzzy logic paper and the piecewise fuzzy set representation introduced by van den Broek.
2. Show how this may be extended in to a general framework for dealing with multiple representations of fuzzy sets. Further we have discussed the possibility for multiple representation to be used at the same time in one fuzzy system, and through implementation we shall demonstrate this working.
3. Finally allow a user of the implemented modules to specify their own representation and co-domain operators. This goal encapsulates the need for providing an abstraction between representation and operators.

We have also noted that this project intends to demonstrate that there are features of the functional programming paradigm and the Haskell language which are advantageous when compared to other languages. To support this argument the following properties have been identified along with where they shall be of use, they also form goals of the implementation:

1. The use of partial application and higher order functions. With reference to van den Broek's paper is was demonstrated that the *apply* function could be partially applied to a representation of a fuzzy set such that a membership function (one from a double value to a double value) was returned.
2. Great potential for polymorphism by virtue of type classes has been discussed, thus we intend in the implementation to show that by taking advantage of this a fuzzy system can be defined and its underlying fuzzy set representation modified with the minimum of effort. This shall be made apparent by both amount of physical code which has to be changed and how localised the placement of these changes is.
3. Due to the closeness to standard mathematical notations the code presented should look more appealing and natural to someone familiar with fuzzy logic operations in general; as covered by an example in the previous section.
4. Code modularisation was covered and we noted that due to the declarative nature of the functional programming paradigm there should be less restrictiveness in the manner in which users of the implementation extend it and integrate it with their existing systems.

Chapter 4

Design methodology

4.1 Summary of chapter

It is the intention of this chapter to provide a clear and rigorous definition of the approach and procedure which are to be used in Chapter 5 when implementations of those goals discussed in Section 3.4 are given. Firstly we address the two major design areas of the project: The parameters of the fuzzy system and the representations to be covered. Secondly we define the procedure we will go through to fully implement those parameters and features identified in the first half.

4.2 Fuzzy design parameters

As outlined in the discussion chapter we shall be building upon the two papers reviewed in Section 2.3, thus our implementation shall provide at least all the fuzzy inferencing functionality offered by the systems presented in those papers. The exact functionality shall be:

Parametric definitions of fuzzy sets using linguistic terms such as *tri(angle)* and *trap(ezoid)* are the standard way to define continuous fuzzy sets. Accordingly we wish to provide such functionality and shall firstly implement *up*, *down* and *tri* functions mimicking the behaviour of the functions of the same names in Meehan and Joy's paper.

In demonstrating the addition of new operators in the final section of the project (as to be covered in the Procedure section) we shall also see a *singleton* fuzzy set parametric definition.

Fuzzy set operators which allow the conjunction or disjunction of two fuzzy sets to be performed using a t-norm or t-conorm [1] function which takes two co-domain values and returns a co-domain value in the returned set.

The following operators shall be shown:

- Zadeh's *maximum* and *minimum.*
- The *product* t-norm
- The Łukasiewicz dual t-norm and t-conorm.

As discussed in Section 3 an objective of the implementation is to demonstrate how co-domain operators may be abstracted from the specific fuzzy set representation used. Thus implementation of these operators shall be a combination of a *junction* function to handle representation abstraction and:

- The standard Haskell functions the operators *max*, *min* and $(*)$.
- Two provided functions *tLuk* and *sLuk* for the Łukasiewicz dual t-norm and t-conorm.

Implication is required and will comprise of a function which takes one of the co-domain operators just covered, a rule firing strength of type *Double* (as defined in Section 2.2.2), a consequent fuzzy set and deliver a resultant set for aggregation. This will be an extension of the $\Rightarrow$ function provided by Meehan and Joy called *implies* which abstracts the operator used (in contrast to their implementation which restricts it to the product operator).

Defuzzification functions will be required to take a fuzzy set and return a domain value as a *Double*. The following will be provided:

- Centroid.
- Minimum, median and maximum of maximum.

A fuzzy inferencing system based on the Mamdani approach covered in Section 2.2.3 shall be demonstrated. To provide this we shall see how a *shoe_size* function similar to the one presented by Meehan and Joy in [21] may be defined to process a list of rules. This shall utilise the fuzzy set operators and implication function which we have just covered.

[1]In fact it shall not be our concern to ensure that functions used obey the properties of a t-(co)norm. We just require them to be of type $Double \rightarrow Double \rightarrow Double$.

4.3 Fuzzy set representations

It is the intention of the project to demonstrate how the features of the function programming paradigm assist the abstraction of the underlying fuzzy set representation. In Section 3 we observed that an appropriate way to do this is using type classes and that this would require the definition of a **class** in Haskell which encapsulates what a fuzzy representation should provide; we shall call this the *FuzzyRep* class.

The functions which a type of representation must provide (that is, a type which is an **instance** of the *FuzzyRep* class) can be identified easily by inspecting the previous section where we specified the parameters for them. They are listed again here with the function names to provide a clear definition of what each representation shall provide. The type variable a is used to indicate a representation type.

Parametric membership functions consisting of either *up*, *down* and *tri* of type $Double \to Double \to a$ or in the discrete case *singleton* of type $Double \to a$.

An application function called *apply* which mimics the behaviour of the function of the same name given by van den Broek. His implementation takes the piecewise representation and returns a membership function, so by abstracting it to any underlying representation we obtain its type as $a \to Double \to Double$.

Fuzzy set manipulating functions called *junction* of type $(Double \to Double) \to a \to a \to a$ and *implies* of type $(Double \to Double) \to Double \to a \to a$. Here the $(Double \to Double)$ functions here are the t-(co)norms used to combine the sets.

An aggregation function which extends the *rulebase* function presented by Meehan and Joy called *aggregate* and of type $(Double \to Double) \to [a] \to a$. From the type we see it takes a t-conorm to use and a list of consequent fuzzy sets to aggregate. Because its definition can be made in terms of a folding operation over *junction* we see that it may be defined as a default function for the class *FuzzyRep* and so individual representations shall not provide their own implementation.

Defuzzification functions are the final component of the *FuzzyG* class, in line with the previous section there shall be four functions: *centroid*, along with *som*, *lom* and *mom* for minimum, maximum and median of maximum respectively. The implementations will be representation specific, however because the median of maximum function can be defined in terms of the minimum and maximum of maximum operations we shall see three implementations per representation and one default function for the median.

To clarify the relationship between these different components and the functionality they provide Figure 4.1 is provided. Here we see how the parametric definitions maybe used to create fuzzy sets with different underlying representations. How two (or one and a firing strength) fuzzy sets may then be used in junction (or implication) operations which also require a fuzzy operator (such as *min*). Finally we see the defuzzifiers each of which takes a single fuzzy set. Having all these components we may then produce a Mamdani fuzzy system.

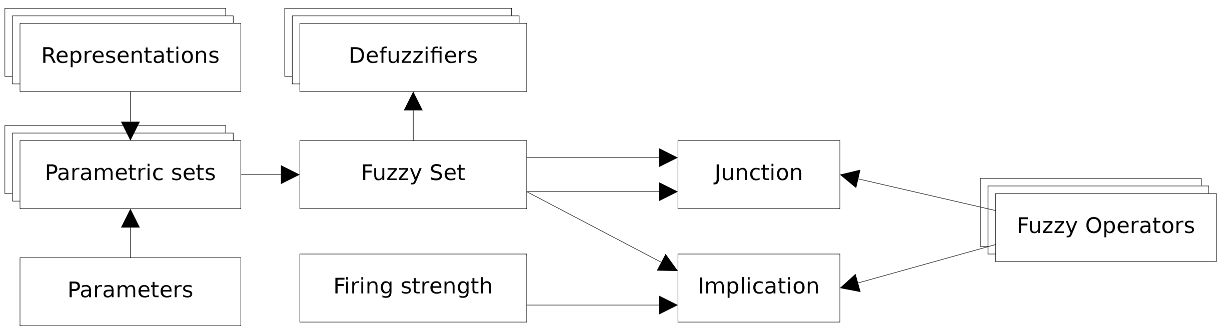

Figure 4.1: Fuzzy logic components and their relationships

4.4 Procedure

Now we have surveyed exactly what functionality is to be provided by the implementation a procedure shall be outlined. Additionally here the module names to be used are defined, these will used in the Haskell code following the module system reviewed in Section 2.4.2. In reading the following it shall be useful to refer to Figure 4.2 where the procedure moves downwards such that the modules to be implemented first are at the top and last at the bottom; the dashed line box highlights the work to be undertaken.

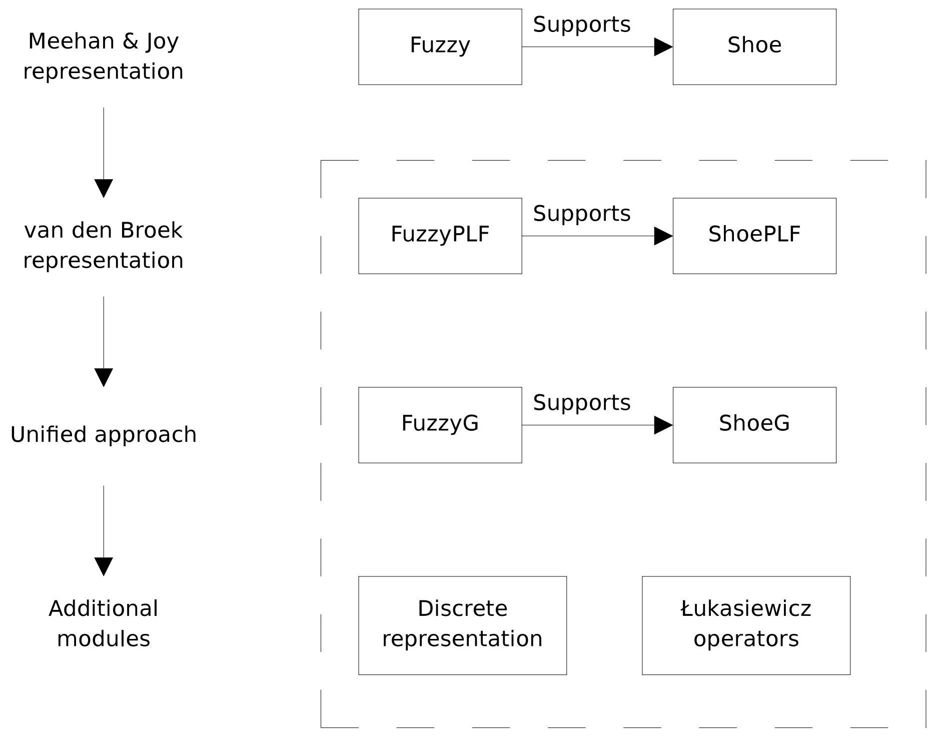

Figure 4.2: Module implementation procedure

1. As can be seen in the diagram the first implementation objective shall be to provide a version of Meehan and Joy's module *Fuzzy* which utilises the piecewise linear representation proposed by van den Broek, this is to be named *FuzzyPLF*. In doing this there are two concerns:

    - Minimising the difference in the type and purpose of those functions exported by the module.
    - Maintaining the performance advantages offered by the van den Broek representation.

    With regard to the former the degree to which this has been achieved will become apparent with the implementation of a version of the *Shoe* module to utilise the functions exported by *FuzzyPLF*, this accordingly shall be called *ShoePLF*. For this first stage to be deemed a success we require there to be a minimal difference in the code implementing *Shoe* and *ShoePLF* and where differences occur we shall critically evaluate why they are necessary.

    For the latter concern an analysis of the system's performance using the Hugs interpreter will form part of the system testing. Here we will evaluate a number of different heights in the range $[1.5, 1.95]$ with the new *shoe_size* fuzzy system and compare them to the result obtained with Meehan and Joy's implementation.

2. The next objective will be the implementation of a system which allows both Meehan and Joy's original representation (simply a membership function mapping domain to degree of membership) and the piecewise linear representation given in the previous objective to be used simultaneously. To facilitate this we see the introduction of the *FuzzyG* class and further analysis of the type signatures and functions it provides (covering those from Section 4.3).

With the *FuzzyG* class defined two instances of it are provided supporting both a modified version of Meehan and Joy's representation and the piecewise representation developed in the *FuzzyPLF* module. The reasons for extending the former are presented during the implementation.

Looking at Figure 4.2 we see *Shoe* module is extended once again to create a module *ShoeG*, this shall provide a revised implementation of the *shoe_size* fuzzy system which utilises the *FuzzyG* module.

Our concerns for this stage are:

- Ensuring that both component representations implemented as **instance** *FuzzyRep* are fully functioning and return the correct values when used with the *ShoeG* module.
- Allow the two representations to be used interchangeably requiring only a type signature to be changed. Further demonstrate that two different representations may be used at the same time following the separation of α and β components covered in the Discussion section.

3. Finally to provide a basis for a critical review of the suitability of the work for use as releasable Haskell fuzzy logic module we must investigate the two features identified in discussion:

   - The ease with which a new representation may be introduced by making it an **instance** of the imported *FuzzyG* class.
   - Modification of the *shoe_size* fuzzy system to use new co-domain operators.

   To do this we shall see implementations of both, for the former a discrete fuzzy set representation and for the latter the Łukasiewicz dual t-norm and t-conorm. Because the concern of this section is to assess suitability as Haskell modules we shall review whether the two implementations may be made without modifying the *FuzzyG* module and discuss the design merits and flaws of the approach taken with regard to module reuse.

Chapter 5

Implementations

5.1 Summary of chapter

Following the procedure established in Section 4.4 of the Methodology chapter we now go through the implementation of the Haskell code which supports the arguments made in the Discussion chapter. Following each of the three steps a review is provided in which the preceding work is critically evaluated, these sections will form the basis for a subsequent review the entire project which is presented in Section 6.2.

5.2 Adaptation of van den Broek representation

5.2.1 The FuzzyPLF module

Here we see how the piecewise linear function approach to representing a membership function (as introduced by van den Broek) may be unified with the *Logic* class approach given by Meehan and Joy in Animated Fuzzy Logic [21]. In the latter paper the module name *Fuzzy* was chosen, thus to indicate the difference this module shall be called *FuzzyPLF*. To demonstrate the use of the *Fuzzy* module an example program residing in the module *Shoe* was given and accordingly our version shall be called *ShoePLF*, its implementation may be found in Section 5.2.2.

```
type Point = (Double, Double)
type PLF = [Point]

instance Logic PLF where
    true = [(0, 1), (1, 1)]
    false = [(0, 0), (1, 0)]
    (∧) = junction min
    (∨) = junction max
    ¬ a = [(x, 1 − y) | (x, y) ← a]
    a ==> c = [(x, y ∗ a) | (x, y) ← c]
```

Let's review the implementations which define our new *PLF* type as an instance of the *Logic* class.

- The definitions for *true* and *false* introduces a restriction not present in the **instance** *Logic Double*, namely a domain must be imposed at this stage such that start and ending domain values can be chosen. Here the domain is specified as the $[0, 1]$ range. Whether this restriction should be considered a disadvantage is questionable when one considers that the *Double* implementation shall impose a domain range when discretisation takes place in the defuzzification step.

- Conjunction and disjunction of piecewise linear fuzzy sets is performed by the abstracted *junction* function which is covered below. It can be seen here that it takes an operator which evaluates two co-domain values, thus the operator is expected to be a t-norm or t-conorm. In mimicking the behaviour of the original paper we can see Zadeh's minimum and maximum operators are used.

- To get a Zadeh negation (as used in the Meehan and Joy paper) of the function we take one minus each co-domain value. Because the function is linear in between these points the one minus will hold at any point.

Next we look at the *junction* function. It and its supporting functions give the functionality provided by the Miranda functions *dac*, *maxplf* and *minplf* in the van den Broek papers [22] covered in Chapter 2. The technique used by *junction* differs slightly in such a way as to provide more what the author regards more readable code, as with the original implementation it relies on the fact that in taking a junction of two fuzzy sets the given t-(co)norm function is applied only to the co-domain values. The critical difference is that van den Broek's implementation moves through both lists (representing the PLFs) at the same time, generating new domain values where necessary and computing the corresponding co-domain values at the same time. In contrast the Haskell implementation given here computes two compatible PLFs (*x1″* and *x2″*) and then combines them using a list comprehension.

```
junction :: (Double → Double → Double) → PLF → PLF → PLF
junction f x1 x2 = [(x, f a b) | ((x, a), (_, b)) ← zip x1'' x2'']
    where
        x1' = discretise x1 x2
        x2' = discretise x2 x1
        x1'' = xovers x1' x2'
        x2'' = xovers x2' x1'
```

The term compatible PLFs is used to denote two which have identical sets of domain values [1] such that at no point in a linear interval between two domain values does the greater co-domain change from one PLF to the other. Formally we state that for a common indexed set of domain values X and two PLFs with membership functions f_1 and f_2 we must satisfy equation 5.1

$$\forall x_n, x_{n+1} \in X, \forall x', x'' \in [x_n, x_{n+1}], i(f_1(x') < f_2(x') \wedge f_1(x') > f_2(x'')) \tag{5.1}$$

To implement this two functions are used to make two PLFs compatible, they are *discretise* and *xovers*. The former ensures the PLF given as the first argument contains all the domain values the PLF in the second argument has and the latter identifies cross over points:

1. Below we see a list comprehension is used to compute all the extra domain and co-domain pairs (those domain values which are present in the second PLF (*plf2*) but not the first (*plf1*). It can be read as:

   Return pairings of x and its co-domain value in the first PLF (*apply x1 x*), where the xs are the domain values of the second PLF (*map fst x2*) which are not domain values in the first PLF ¬ $ *elem x* (*map fst x1*). Because all extra domain and co-domain pairs are appended (++) to the end of the first PLF the list must be sorted by the *orderpoints* function before being returned.

   ```
   discretise :: PLF → PLF → PLF
   discretise x1 x2 = orderpoints $ x1 ++ [(x, apply x1 x) | x ← map fst x2, ¬ $ elem x (map fst x1)]
   ```

2. Checking all the cross over points requires that both input PLFs have identical sets of domain points as guaranteed by the *discretise* function. With this true one pairs up the points in each PLF to create two lists of segments, the corresponding segments in each PLF are then inspected with the *xover* function to see if they cross.

   ```
   xovers :: PLF → PLF → PLF
   xovers x1 x2 = orderpoints $ x1 ++ (catMaybes $ map xover $ zip (pairs x1) (pairs x2))
   ```

   Below *xover* checks if the lines given by two segments cross over each other. If not (such that all co-domain values in the second line are either above or are all below the first line) then *Nothing* is returned. In the case that they do *Just* the point at which they cross is returned. The *catMaybes* function in *xovers* can then eliminate those segments which do not contain a cross and append those which do.

   ```
   xover :: ((Point, Point), (Point, Point)) → Maybe Point
   xover (((x0, a0), (x1, a1)), ((_, b0), (_, b1)))
       | (a1 − a0) ≡ (b1 − b0) = Nothing
       | (x0 ⩾ y) ∨ (y ⩾ x1) = Nothing
       | otherwise = Just (y, c)
       where
           y = x0 + (x1 − x0) ∗ (a0 − b0) / (a0 − b0 + b1 − a1)
           c = a0 + (a1 − a0) ∗ (y − x0) / (x1 − x0)
   ```

The *apply* function shown below is a direct Haskell implementation of the Miranda function of the same name and serving the same purpose in van den Broek's paper. We shall see how it is unified with the Meehan and Joy system in the module *ShoePLF*.

[1]Haskell's list comprehension syntax requires the more correct expression $((x, a), (x, b))$ to be written in the implementation as $((x, a), (\_, b))$ in the *junction* function.

```
apply :: PLF → Double → Double
apply [ ] _ = 0
apply [_] _ = 0
apply ((x0, y0) : (x1, y1) : zs) x
    | x < x0 = 0
    | (x0 ≡ x1) ∨ (x > x1) = apply ((x1, y1) : zs) x
    | otherwise = y0 + (x − x0) ∗ (y1 − y0) / (x1 − x0)
```

Piecewise linear versions of the *up*, *down* and *tri* functions are implemented using the van den Broek representation. One notable deviation from the Meehan and Joy versions are that domain values outside the $[a, b]$ are undefined with these representations. To alleviate this problem the *apply* function can be seen to return a membership of grade of 0 for any domain value which is not covered by the representation.

```
up, down, tri :: Double → Double → PLF

up a b = [(a, 0), (b, 1)]
down a b = [(a, 1), (b, 0)]
tri a b = [(a, 0), (mid, 1), (b, 0)]
  where mid = a + ((b − a) / 2)
```

The four defuzzification techniques shown in "Animated Fuzzy Logic" are now implemented for van den Broek representation. Here we see the advantage of the representation become apparent as the values are computed numerically rather than relying on discretisation; this topic was covered in Section 2.2.3.

Due to the nature of the algorithm used for the *centroid* function the piecewise function must be *bound* at both ends. This can be seen to find the *minimum* and *maximum* domain values (i.e. the start and end points of the function) and create new co-domain values of zero for these. The general recursion scheme provided by *foldr2* processes each segment of the piecewise function in turn (provided by the repeated application of tail implemented as *iterate tail*) until all segments have been computed (and only one domain / co-domain pair remains as identified by (>1) ∘ *length*).

```
centroid :: PLF → Double
centroid plf = (1 / (6 ∗ (0.5 ∗ (sum $ area $ bound plf)))) ∗ (sum $ cent $ bound plf)
    where
        bound xs = [(minimum $ map fst xs, 0)] ++ xs ++ [(maximum $ map fst xs, 0)]
        cent = foldr2 (λ((x0, y0) : (x1, y1) : _) → (x0 + x1) ∗ ((x0 ∗ y1) − (x1 ∗ y0)))
        area = foldr2 (λ((x0, y0) : (x1, y1) : _) → ((x0 ∗ y1) − (x1 ∗ y0)))
        foldr2 f xs = map f ∘ takeWhile ((>1) ∘ length) ∘ iterate tail $ xs
```

The median of maximum defuzzification technique is trivially defined as a function of the other two.

```
minmax, medmax, maxmax :: PLF → Double
medmax plf = (minmax plf + maxmax plf) / 2
```

We can see *minmax* and *maxmax* utilise the list comprehension covered in the review of Haskell syntax in Section 2.4 and described there.

```
minmax plf = minimum [x | (x, y) ← plf, y ⩾ maxy]
    where maxy = maximum (map snd plf)
maxmax plf = maximum [x | (x, y) ← plf, y ⩾ maxy]
    where maxy = maximum (map snd plf)
```

Notably the *rulebase* function is identical to the definition in "Animated Fuzzy Logic", this is due to ad-hoc polymorphism as covered in Section 2.4.

```
rulebase :: Logic a ⇒ (a → a → a) → [a] → a
rulebase = foldr1
```

5.2.2 The ShoePLF module

In this module we see the adaptations which are required to be made to Meehan and Joy's *Shoe* module such that it works with the author's *FuzzyPLF* module. The first observation to be made is that the *sizes* :: *Domain ShoeSize* definition is no longer needed. This served as a list of domain values to be used by the defuzzification function as covered in the review in Section 2.3, however because van den Broek's piecewise linear function approach removes the need for discretization during defuzzification (as covered in the review and in Section 2.2.3) the list is no longer required.

Below we see the definitions of the membership functions differ only in their type signatures, changing from *Fuzzy Height* and *Fuzzy ShoeSize* (which recall are type synonym for *Double* → *Double* functions) to *PLF* (a synonym of [(Double, Double)]. Despite this small change it is important to note the fundamental change from an actual membership function to a *representation* of a membership function. We shall see in the *shoe_size* function below how this is addressed.

```
short, medium, tall, very_tall :: PLF
short = down 1.5 1.625
medium = tri 1.525 1.775
tall = tri 1.675 1.925
very_tall = up 1.825 1.95

small, average, big, very_big :: PLF
small = down 4 6
average = tri 5 9
big = tri 8 12
very_big = up 11 13
```

Three changes must be made to the *shoe_size* function given in the Meehan and Joy paper to make it compatible with the author's *FuzzyPLF* module:

1. As identified in the previous paragraph an issue exists using the van den Broek representation because unlike the approach taken by Meehan and Joy it is not actually executable in the sense that it is no longer a function from domain to co-domain value. A solution to this is provided by way of the *apply* function (of type *PLF* → *Double* → *Double* and we can see below how by prefixing the membership function representations with this function they evaluate to a *Double*, satisfying the type of the first argument of the implication operator ⇒.

2. Because the PLF version of the *centroid* function no longer needs a domain value it can be eliminated as an argument.

3. Lastly our change in approach sees the use of the *junction* function. This contrasts with the previous approach in which Haskell's type class system was used to allow the addition of fuzzy sets by defining how the + function should operate over fuzzy sets (that is + overloaded to the type (*Double* → *Double*) → (*Double* → *Double*) → (*Double* → *Double*)). In the next section we shall investigate the advantages and disadvantages of these two approaches with regard to the author's generalised *shoe_size* function given in Section 5.3.3.

```
shoe_size :: Height → ShoeSize
shoe_size h = centroid (
    rulebase (junction (+)) [
    apply short h ⇒ small,
    apply medium h ⇒ average,
    apply tall h ⇒ big,
    apply very_tall h ⇒ very_big])
```

5.2.3 Review of implementation

Having shown that it is possible to adapt the representation suggested by van den Broek to fit the fuzzy inferencing system put forward by Meehan and Joy (as an implementation of the *shoe_size* function) we see a critical evaluation of the pros and cons of the implementation. In conclusion we review what has been learned from the experience in preparation for the next step specified in our design methodology procedure in Section 4.4.

Module	Dom	Height										Work
		1.5	1.55	1.6	1.65	1.7	1.75	1.8	1.85	1.9	1.95	
Shoe	10	4.3	5.15	6.47	7.0	7.75	9.25	10.0	10.53	11.85	12.67	13,903
Shoe	50	4.61	5.51	6.63	7.0	7.75	9.25	10.0	10.37	11.49	12.39	62,431
Shoe	100	4.6	5.56	6.65	7.0	7.75	9.25	10.0	10.35	11.44	12.36	123,041
ShoePLF	N/A	4.67	5.6	6.67	7.0	7.75	9.25	10.0	10.3	11.4	12.3	73,969

Table 5.1: Performance of PLF representation against Meehan and Joy representation.

Accomplishments of this system include firstly that we have successfully demonstrated that van den Broek's representation can be used to implement a fuzzy system. It was tested against Meehan and Joy's module and results are presented in Table 5.2.3; here we see the evaluation of ten shoe sizes in the range $[1.5, 1.95]$ using both *shoe_size* function implementations. Additionally three cardinalities (10, 50 and 100) of discretised domain sets were used for the *Shoe* implementation of the *centroid* function, this allows us to observe two properties:

- The accuracy of the defuzzification process, comparing the sampling of a discretised domain method used by Meehan and Joy in *Fuzzy* to the geometric approach which is possible with van den Broek's representation as implemented in *FuzzyPLF*.
- The computational complexity of performing the ten fuzzy inferencing operations. To gauge this the author has given the reduction count under the work column as reported in a Hugs session, this is covered in Section 2.4.2.

Looking at the results we can see that as the number of sampling points is increased when using the *Shoe* module the defuzzified shoe size values tend towards those computed by the *ShoePLF* module whilst the amount of computation required increases. As such we can assert that the *ShoePLF* and *FuzzyPLF* modules are functioning correctly.

Failings of the system are foremost that the produced *ShoePLF* module's definition of *shoe_size* is not identical to the one in the *Shoe* module, making it uneasy for a user of the system to switch between the two. We surveyed the reasons for the discrepancies alongside the implementation and in line with the requirements set down in the methodology section they can be accepted at this stage of the implementation. A second issue is the *true* and *false* functions as defined by Meehan and Joy which are difficult to unify with a piecewise linear representation with out either performing scaling and un-scaling operations (causing more deviations between the *Shoe* and *ShoePLF* modules) or providing a bounding range as an argument to the functions (changing their types).

Finally one issue still present in the *ShoePLF* module is the "hard coding" of the co-domain operators used in functions such as $\wedge$ and $\Rightarrow$ (using minimum and product respectively). This suggests a deeper limitation in the approach taken by Meehan and Joy where the *Logic* class is used. We observe that to encapsulate multiple operators and to allow a user of the system to specify them without having to alter the underlying *Fuzzy* module we must adapt this approach. We would expect a successful approach to allow operators to be specified during the definition of the fuzzy system (such as in the *Shoe* and *ShoePLF* modules.

In conclusion we briefly review what has been learned during this implementation:

- It is possible to implement a fuzzy system in Haskell which uses a fuzzy representation other than a function from domain value to a degree of membership. To do this we used the *apply* function.
- In their current state a user of the modules is not able to easily switch between representations, to facilitate this we will have to further adapt the underlying framework.
- Whilst we have used Meehan and Joy's *Logic* class in this implementation we see that it specifically is preventing the user from providing their own co-domain operators. However the use of the type class system available in Haskell still has potential; we should investigate adapting the *Logic* class to allow multiple operators.

5.3 Abstraction of representation

5.3.1 Analysis of differences

In deriving an abstraction which shall allow multiple fuzzy set representations to be used we first compare the original Meehan and Joy implementation of the *Shoe* module (which used a *Double* → *Double* representation) to the one presented in Section 5.2 (which used a piecewise linear function representation). With the differences identified the steps and decisions required to unify the two representations are then discussed; following this the implementation is given.

1. The introduction of the *apply* operator to support van den Broek's representation may trivially be unified with the Joy and Meehan representation by observing that when the latter is used we may define *apply* to be the identity function *id* [2]. Looking at the type of the identity function $a \rightarrow a$ and the type of the Meehan and Joy representation *Double* → *Double* we can see that when the latter is applied to the former we again get a membership function of type *Double* → *Double* as required.

   Because the Meehan and Joy representation is to be changed as covered shortly, the final definition of *apply* for the representation shall not be the identity function *id* as given here. The reason for this is covered in line with the implementation.

2. Because defuzzification of a Meehan and Joy representation fuzzy set requires discretisation and the piecewise linear function one does not there exists a situation where the former requires a domain set to be provided [3] and the latter does not. This creates a problem in unifying the types of a defuzzification method where the former is of type *Domain* → *FuzzySet* → *Double* and the latter is just *FuzzySet* → *Double*. The author has identified the following possible solutions:

   - Make the requirement that all defuzzification functions are of type *Domain* → *FuzzySet* → *Double*. In this approach any representation which supports a numerical (rather than discretisation) based algorithm may simply disregard the provided domain.
   - Design a system whereby the domain to be used for discretisation may be derived automatically from the representation. To realise this one must know the range of the domain (over the set of real numbers) and choose a metric to establish how many sampling points to use when performing the discretisation.
   - A third approach would utilise scaling operations. By scaling and un-scaling all domain ranges to the $[0, 1]$ range we could guarantee the all defuzzification functions would only discretise in that range. The number of sampling points to take within this range would again be arbitrary and could be established by some metric.

   After consideration the second option has been chosen for several reasons. Firstly the first approach requires over specification for any representation which gives the advantage of not requiring a domain to be explicitly given. Secondly the range of the domain may be derived implicitly from the definition of the function when parametric membership functions such as the *up* function are used. Finally because we can derive the range of a membership function implicitly in this way the scaling approach again introduces additional constraints on representations which do not require it (i.e. they shall have to be scaled and un-scaled).

   In the following implementation we shall see how the Meehan and Joy representation is adapted to encapsulate the domain range and how the parametric membership functions (such as *up*) and defuzzification functions (such as *centroid*) may be adapted to utilise this.

3. As mentioned in the concluding part of Section 5.2.2 Meehan and Joy had opted to use Haskell's type class system for fuzzy set aggregation where our approach provided the function *junction*. In addressing this difference firstly let us consider the two approaches separately and the impact on the definition of an aggregation operation:

[2] This is defined in Haskell as: *id* $x = x$.

[3] This domain set is either the universal set (as covered in Section 2.2.1) or in the case that the universal set is continuous it is a uniform finite sampling of the universal set.

- From Meehan and Joy's definition of their representation as an instance of the *Num* class they define addition thus: $f + g = \lambda x \rightarrow f\ x + g\ x$. This states that the addition of two fuzzy sets is performed by adding the co-domain values (degrees of membership) for any domain value.
- Our *junction* operation provides similar functionality over the van den Broek representation but abstracts out the specific operation used for aggregation to allow any t-(co)norm function of the type $Double \rightarrow Double \rightarrow Double$ to be used. This abstraction is possible because as noted previously when performing a junction only the co-domain values are compared. As shall be demonstrated during the implementation stage in the next section this is an important property because it allows the t-(co)norm function to be separated from the underlying representation.

As noted in line with the implementation in the previous section because we are no longer using Haskell's type class system and polymorphism select the correct addition function (+) (by making ($Double \rightarrow Double$) an instance of the *Num* class) the function to be used for aggregation must be prefixed with the higher order *junction* function.

Specifically due to the property of allowing a t-(co)norm function to be defined separately from a representation's definition of the *junction* function the author shall be choosing this approach in their implementation. A further discussion of the benefit of this approach and how the functional paradigm allows it to be elegantly expressed is given in the evaluation in Section 6.3.1. A second implication of this change shall be seen in that we may make the application of the t-(co)norm function to the *junction* appear to occur 'automatically' by composing it with an aggregation function.

5.3.2 The FuzzyG module

Encapsulating all the abstractions identified in the previous section the author now introduces the *FuzzyG* [4] module which contains firstly a class definition of what functions a representation must provide and secondly two instantiations of this class which are modifications of both the Joy and Meehan representation and the author's Haskell implementation of the van den Broek representation. An introduction to type classes is given in Section 2.4.1 and the notation used to define them in Haskell is covered in Section 2.4.2, familiarity with the concept is assumed in the following sections.

The FuzzyRep class

Firstly for brevity a type synonym is given for t-norms and t-conorms , it shall be called *TSNorm* and it's type indicates it takes two co-domain (degree of membership) values and returns one as covered in Section 2.2.2. Next we begin the definition of the *FuzzyRep* class.

> **type** $TSNorm = (Double \rightarrow Double \rightarrow Double)$
> **class** *FuzzyRep a* **where**

Now a list of the functions a representation must provide is given, we refer to this as the minimal complete definition for the class *FuzzyRep*. As can be seen in the opening line of the definition above the type variable *a* is the representation type.

With this in mind we firstly inspect the type signature for the parametric membership functions *up*, *down* and *tri*. We observe that the type signatures for these functions in both representations (the first as reviewed in Section 2.3 and the second as implemented in module *FuzzyPLF*) are the same; taking two doubles to indicate the start and end of the range of definition over the domain and returning the fuzzy set. Thus we see the type variable *a* is now used instead of a specific representation.

$$up, down, tri :: Double \rightarrow Double \rightarrow a$$

Next we inspect two more previously seen functions, *apply* and *junction*.

$$apply :: a \rightarrow Double \rightarrow Double$$
$$junction :: TSNorm \rightarrow a \rightarrow a \rightarrow a$$

[4] The name stems from Fuzzy Generic by virtue of it allowing parameterisation of its functions.

We can read the type signature of *apply* as a function which takes the representation a and returns a membership function *Double* → *Double* (by virtue of partial application as covered in Section 2.4.2). The abstraction the *junction* function provides between the t-(co)norm function used and the underlying fuzzy set representation can be seen here in the type signature. We shall see in the following section how each representation may implement *junction* by utilising the t-(co)norm function to compute co-domain values for a resultant set in the representation.

Next the author presents two new functions, the first of which called *implies* provides an abstraction for implication in a similar fashion to *junction* for conjunction and disjunction; its purpose is once again to allow any co-domain operator (such as product) to be used. Differing from *junction* and adhering to the argument types shown diagrammatically in Figure 4.1 it takes the implication operator to use, a *Double* indicating the firing strength of the antecedent and the consequent fuzzy set defined with representation of type a; a resultant fuzzy set with representation of type a is returned.

The second new function is a refactoring of the *rulebase* function used in the two previous *Fuzzy* modules. Because we need the *junction* function to allow different aggregation operators to be used we can wrap the application of it up into the aggregation operation as shown below. Thus the *TSNorm* function is partially applied as the first argument to *junction* which yields a co-domain to co-domain function, the composition operator ∘ is then used to partially apply this function to *foldr1* which folds all the consequent fuzzy sets together.

```
implies :: TSNorm → Double → a → a
aggregate :: TSNorm → [a] → a
aggregate = foldr1 ∘ junction
```

The three defuzzification methods minimum (smalled) of maximum, maximum (largest) of maximum and centroid are given by functions specific to the representation, we list them now:

```
centroid, som, lom :: a → Double
```

Because the median of maximum function is determined in terms of the other two we may implement it here as part of the class definition. Due to polymorphism the compiler can then select the correct implementations of *som* and *lom* to use for the representation a passed in.

```
mom :: a → Double
mom a = ((som a) + (lom a)) / 2
```

Instances of the FuzzyRep class

Having established the criteria for a representation of a fuzzy set in the previous section we now see how the two representations covered so far may be unified with this approach. Firstly the Meehan and Joy representation is covered and secondly van den Broek's piecewise linear function representation.

Following the order adopted in the definition of the class firstly implementations of the parameterised membership functions are given, followed by the inferencing functions *apply*, *junction*, *implies* and lastly the defuzzification functions.

Meehan and Joy representation. Before presentation of the implementation of the *FuzzyRep* class for this representation we must address the decision made in the previous section to include a range of domain values such that discretisation can be performed automatically.To realise this the representation shall be extended to compose of both the membership function as the previous approach did and also a new component indicating the range of valid domain values. Thus the type of this new representation shall be a tuple containing firstly the membership function (*Double* → *Double*) and secondly a tuple indicating the start and ending bounds of the domain range ((*Double*, *Double*)), the name bounded continuous function shall be used to refer to this for which the type synonym *BCF* is defined:

```
type BCF = ((Double → Double), (Double, Double))
```

Having established this new type we now provide definitions for each of the functions required by the minimal complete definition of the *FuzzyRep* class for the specific instance that the type variable a is bound to our new *BCF* representation. Following convention the parametric functions are defined first:

instance *FuzzyRep BCF* **where**

$$up\ a\ b = (up'\ a\ b, (a, b))$$
$$down\ a\ b = (down'\ a\ b, (a, b))$$
$$tri\ a\ b = (tri'\ a\ b, (a, b))$$

These definitions utilise those given in the Meehan and Joy paper and the functions from there are indicated by a prime. We see that using the newer *BCF* representation the given range is also stored in the second part of the returned tuple (a, b).

Next we cover the *apply*, *junction* and *implies* functions:

$$apply = fst$$

In Section 5.3.1 the notion that for the Meehan and Joy representation *apply* was defined as the identity function *id* was covered. Because this has now been extended to the bounded continuous representation we use the function *fst*, this is defined as $fst\ (a, b) = a$ and as such we can see it returns the membership function component of the representations membership function / range tuple. For the next two function the $(a, (p, q))$ notation is used, this unpacks the representation tuple and binds the membership function component to the a variable, the start of the domain of definition to p and the end to q.

```
junction f (a, (p, q)) (b, (r, s)) = (λx → f (a x) (b x), (t, u))
    where
        t = min p r
        u = max q s
```

We can see here that the first component of the returned tuple $(\lambda x \rightarrow f\ (a\ x)\ (b\ x))$ of *junction* is reminiscent of its early incarnation in the Meehan and Joy paper $(\lambda x \rightarrow f\ x + g\ x)$ where it defined addition in making the representation an instance of the *Num* class. Additionally because the new representation requires the domain of the fuzzy set to be maintained some book keeping is performed; we simply compute a new range (t, u) which guarantees to contain both the component fuzzy sets domain's.

Next the *implies* function is defined, this again is derived from the Meehan and Joy implementation where they used the infix ⇒ operator thus: $w \Rightarrow f = \lambda x \rightarrow w \Rightarrow f\ x$. Again we see the difference between the reliance on the type class system to resolve polymorphism which Meehan and Joy chose to fix as the product operator with the definition ⇒ for the *Double* instance. In contrast the author's approach allows any function f to be used as an implication operator, in Section 5.3.3 we shall see how this approach fits into the original definition of the *shoe_size* function.

$$implies\ f\ s\ (a, r) = (\lambda x \rightarrow f\ s\ (a\ x), r)$$

This concludes the minimal definitions for the abstract functions of the bounded continuous function representation and accordingly we now inspect the defuzzification methods implemented. As discussed previously the new bounded representation allows the discretised domain to be computed automatically from the range defined by the second element of the tuple. To allow Meehan and Joy's implementations of the defuzzification functions to be unified with this approach with minimal effort we define a function *dom* [5] to perform this automatic discretisation as follows, this definition uses a list syntax covered in Section 2.4.2.

$$dom\ (p, q) = [p, p + (q - p) / 99 \mathinner{..} q]$$

So we can read this as a function which takes a start and end point and returns a list which always has a length of one hundred elements where the first is the start of the domain, the last is the end of the domain and the remaining eight are uniformly distributed between them. The value of one hundred was chosen arbitrarily.

[5] Become the function *dom* does not form part of the complete minimal definition for the *FuzzyRep* class we must define it outside the declaration. Thus it is actually located below the **instance** block.

Now we see how Meehan and Joy's *centroid* function is adapted to unify with this approach. To do this there are two tasks: Generate a discretised domain using the *dom* function and then use the original algorithm. The generation of the discretised domain can be performed in place by substituting occurrences of *dom* in the original definition with the evaluation *dom r* where *r* can be seen to contain the range tuple.

```
centroid (f, r) = (sum (zipWith (*) (dom r) fdom)) / (sum fdom)
    where fdom = map f (dom r)
```

Piecewise linear function approach. Now we see how the function derived in for the *FuzzyPLF* module in Section 5.2.1 may be unified with the *FuzzyRep* class.

```
instance FuzzyRep PLF where
```

Firstly the parametric definitions of fuzzy sets given by *up*, *down* and *tri* are defined for the *PLF* instance of the *FuzzyRep* class. Due to preservation of the initial naming scheme and the identical types as noted in the class definition one may simply use the definitions provided in the *FuzzyPLF* module verbatim.

Again having already provided implementations of three of the functions (*apply*, *junction* and *centroid*) required to unify van den Broek's representation with Meehan and Joy's approach they shall not be given here. The fourth function required by the minimal definition of the *FuzzyRep* class is *implies* and this is also implemented in *FuzzyPLF* as $\Rightarrow$ but fixed to use the product implication operation. Here we provide an implementation which abstracts the operator used to the argument f, in Section 5.3.3 we see how partial application is used to obtain the original $\Rightarrow$ function by defining it equal to *implies* $(*)$.

```
implies f s a = [(x, f s y) | (x, y) ← a]
```

5.3.3 The ShoeG module

As per the procedure outlined in the methodology chapter we now see a third version of the shoe size fuzzy system originally presented by Meehan and Joy. We can see the first two lines are identical to the previous versions defining the module name and importing the module with the fuzzy operations in.

```
module ShoeG where
import FuzzyG
```

In discussion in Chapter 3 we stated that we would show a fuzzy system which is capable of using multiple representations at the same time and that feature is shown now:

```
short = down 1.5 1.625 :: BCF
medium = tri 1.525 1.775 :: BCF
tall = tri 1.675 1.925 :: BCF
very_tall = up 1.825 1.95 :: BCF

small = down 4 6 :: PLF
average = tri 5 9 :: PLF
big = tri 8 12 :: PLF
very_big = up 11 13 :: PLF
```

Here we see the use of the parametric membership functions as before in both *Shoe* and *ShoePLF*. The notable difference here is the addition of a type signature to each line, this allows the compiler to identify which underlying representation to use (and thus which instance of the parametric function to evaluate the two parameters with). For the reasons outlined in the discussion chapter we observe that only a maximum of two fuzzy set representations [6] may be used this Mamdani system and we can see the *BCF* representation chosen for α and *PLF* for β.

[6]Recall in the discussion we name these two representations α and β, the former is the antecedent fuzzy sets and the latter the consequent ones.

Next to mimic the behaviour of the original *Shoe* module we see how the new *implies* function is partially applied with the product operator $(*)$. This creates a function which we can prove is identical to the *Shoe* function $\Rightarrow$. Firstly we see the definition and then a short proof demonstrates how it is identical:

```
infix 0 ⇒
(⇒) :: FuzzyRep a ⇒ Double → a → a
(⇒) = implies (∗)
```

For this proof we required to demonstrate that the membership function component (*fst*) of *implies* $(*)$ is identical to the $\Rightarrow$ from *Shoe*, to do this we start with a definition of the former and derive a definition of the latter.

$fst\ (implies\ (*)\ s\ (a, r)) =$	$\lambda x \rightarrow (*)\ s\ (a\ x)$	
		from definition of implies
	$\lambda x \rightarrow s * (a\ x)$	
		moving $(*)$ *to infix*
	$\lambda x \rightarrow s \Rightarrow (a\ x)$	
		from definition of $(\Rightarrow)$ *for* **instance** *Logic Double*
$s \Rightarrow a =$	$\lambda x \rightarrow s \Rightarrow (a\ x)$	
		from definition of $(\Rightarrow)$ *for* **instance** $(Logic\ b) \Rightarrow Logic\ (a \rightarrow b)$

Finally we see the definition of the *shoe_size* function using our new generic representation approach. As with the *FuzzyPLF* module and for the reason identified during the discussion chapter the *apply* function is still required. Finally we see the new *aggregate* function wraps up the *rulebase* and *junction* functions, if we wished we could further emulate the original definition of *shoe_size* given in Meehan and Joy's *Shoe* module using using partial application as with $(\Rightarrow)$ thus: $rulebase = aggregate\ (+)$.

```
shoe_size :: Double → Double
shoe_size h = centroid (
    aggregate (+) [
    apply short h ⇒ small,
    apply medium h ⇒ average,
    apply tall h ⇒ big,
    apply very_tall h ⇒ very_big])
```

5.3.4 Review of implementation

In this section we have seen the development of a successor to Meehan and Joy's *Logic* class which supports multiple representations, *FuzzyRep*. Again we now look at the pros and cons of this approach to provide a basis for the evaluation chapter of the report.

Accomplishments of the implemented system in this chapter are apparent from inspecting the *ShoeG* module. Here we see two different underlying representations (BCMFs and PLMFs) being used simultaneously by specifying their type explicitly using a type signature. As was suggested before undertaking the implementation we have seen how Haskell's type class system is then used to allow the compiler to determine the representation specific functions to use.

Further the new *FuzzyRep* class allows the use of user specified co-domain operators as can be seen in the expressions *implies* $(*)$ and *aggregate* $(+)$. This is an important step towards the goal of providing usable fuzzy logic modules for Haskell for the reasons identified in the discussion chapter.

As with the *PLF* implementation in the previous section the new *shoe_size* function has been tested against a range of ten heights, the results are given in Table 5.3.4 and can be seen to be correct when compared to the original *Shoe* module.

Module	Dom	Height									
		1.5	1.55	1.6	1.65	1.7	1.75	1.8	1.85	1.9	1.95
Shoe	100	4.6	5.56	6.65	7.0	7.75	9.25	10.0	10.35	11.44	12.36
ShoeG	100	4.67	5.60	6.67	7.0	7.75	9.25	10.0	10.33	11.40	12.33

Table 5.2: Testing of *ShoeG* module.

Failings of this system are two fold. Firstly whilst a step towards modularisation has been taken the restriction of defining a class all in one place makes it impossible for a user to add new parametric membership functions or defuzzification methods without altering the *FuzzyG* module. This problem will be demonstrated in the next section where we see what would be required to extend the system in this way.

Secondly a questionable property of the new *BCF* representation is the fixing of a discretised set cardinality to 100. As noted this was an arbitrary choice but critically it again requires the user to manually modify the *FuzzyG* module, worse still it makes it impossible [7] to have one fuzzy system with different discretisation levels.

Scope for developing solutions to both these problems are put forwards in the evaluation of the system in Section 6.2.3.

Again we observe what has been learned from this implementation:

- Through the use of type classes and polymorphism it is possible to develop a fuzzy system which provides an abstraction of fuzzy set representation.
- However in its current form there are still too many restrictions to consider the modules usable for an end user. We have enumerates these restrictions in in the previous paragraphs.
- The α and β representation concept which showed that two different ones could be used in a single fuzzy system (as covered in Section 3.3) was shown to be viable through implementation.

5.4 Demonstration of new approach

5.4.1 A discrete representation

To demonstrate how the *FuzzyG* class maybe extended to support new representations we now provide an implementation of a discrete fuzzy set representation according to the definition given in Section 2.2.1. To show how this does not rely on the original *FuzzyG* module we place it in its own module called *Discrete*, the *FuzzyG* module is then imported bringing the definition of the *FuzzyRep* class in to scope.

```
module Discrete where
import FuzzyG
```

A natural representation for a discrete fuzzy set is an ordered list of domain value / degree of membership pairs, exactly as the standard notation given in equation 2.3. However this creates a problem because we have already defined the Haskell type [(*Double*, *Double*)] to be synonymous with the *PLF* representation.

Our solution to this problem is to create a new algebraic data type (covered in Sections 2.4.1 and 2.4.2) using the Haskell **newtype** key word, we call the new type *DF* for discrete function. As previously we then provide an instance definition for our new representation.

```
newtype DF = DF [(Double, Double)] deriving Show
instance FuzzyRep DF where
```

[7]This is not entirely true, but the solution would require one to make an identical copy of the *FuzzyG* module, for example *FuzzyG2*, change 100 to the required value and then use qualified importing to manage which version to use. Clearly not an appropriate solution.

Immediately we encounter another problem because the three parametric membership function constructors (*up*, *down* and *tri*) we defined in the minimal definition of *FuzzyRep* are undefined for a discrete set [8]. This is not a problem in Haskell because the compiler by default places no demand that an instance of a class provides implementations of all the functions in the minimal definition. Thus we may omit them, we discuss the effect of this when we review the implementation in Section 5.4.4.

Without these three definitions it would be convenient to provide a parametric definition appropriate to a discrete fuzzy set, for this we choose the singleton fuzzy set which has one double value with a degree of membership of 1, and 0 for all other double values. We note at this time that no *singleton* function is defined in the minimal definition for the class *FuzzyRep* and as such it cannot be present as part of the declaration **instance** *FuzzyRep DF* unless it is added first. The ramifications of this are discussed further in the review Section 5.4.4.

The definition we provide below is thus given out side the scope of the instance declaration, this causes no problem because the function name *singleton* is not defined for any other representation type [9].

```
singleton x = DF [(x, 1)]
```

We see above the trivial definition, indicating that the tuple for a fuzzy set A is of the type $(x, \mu_A(x))$. We can also see the type constructor *DF* being used. Next the apply function is defined, for this we can use Haskell's standard list function *lookup* of type $a \to [(a, b)] \to$ *Maybe* b. This function attempts to find a key (of type a) in a list of key / value pairs (represented as (a, b) tuples) and return the corresponding value (of type b). To deal with the possibility of not finding the key in the given list it used the *Maybe* type to return either *Just* the corresponding value if the key was found or *Nothing* if it was not.

Looking at the representation we are using for a discrete fuzzy set we can see the types match up nicely where the key of type a is the domain value (a *Double*) and the corresponding value (b) is its degree of membership to the set (also a *Double*). Below we see a **case** statement is used to deal with the two possibilities that either a domain value is defined in the set (and *Just* its degree of membership is returned) or it isn't (and *Nothing* is returned).

```
apply (DF a) x = case (lookup x a) of
    Just ax → ax
    Nothing → 0
```

It the evaluation of the *lookup x a* matches the *Just* case we extract the degree of membership (ax) and return it. Alternatively if the evaluation returns *Nothing* then we return a degree of membership of zero.

Next we see how this *apply* function can be used to provide a concise definition of *junction* as shown below.

```
junction f (DF a) (DF b) = let
    dom = union (map fst a) (map fst b)
    ab = [(x, f (apply (DF a) x) (apply (DF b) x)) | x ← dom]
    in DF (filter ((>0) ∘ snd) ab)
```

Inspecting the first line we again see the *DF* type constructor used to expose the list representations for the two discrete fuzzy sets a and b. Then follows three components of the solution:

1. Firstly we compute the domain of the resultant set *dom*, this is a crisp logic *union* of the domain sets (extracted by *map fst*) of each list.

2. Next a list comprehension defines the junction of the two as ab. We can see it operates by taking each domain value from *dom* in turn as x and uses *apply* to obtain the corresponding degree of membership from each fuzzy set. The domain value x is then paired its degree of membership in the resultant set as computed by the f function.

[8]We could if we wished provide definitions which generated a discretised set of values fitting the *up*, *down* and *tri* shape but this is some what counter intuitive when using a discrete set.

[9]Contrast this with *up*, *down* and *tri* which are defined over two representation types.

3. Finally the list *ab* is evaluated against a *filter* which is used to eliminate any domain values in the resultant set which have a degree of membership of zero [10], expressed in Haskell by saying the second tuple component must be greater than zero $((>0) \circ snd)$. The *DF* type constructor is then used to return return the list as a *DF*.

Next we see the *implies* function is an extended version of *implies* for the *PLF* instance of *FuzzyRep*. The additions are the *DF* type constructor which is now required and once again the *filter* to remove domain values which have zero membership to the resultant set.

$$implies\ f\ s\ (DF\ a) = DF\ (filter\ ((>0) \circ snd)\ [(x, f\ s\ y) \mid (x, y) \leftarrow a])$$

Finally we see the implementation of the *centroid* defuzzification method as derived from the mathematical definition given for a fuzzy set A with N defined domain values as:

$$\frac{\sum_{i=0}^{N} x_i \times \mu_A(x_i)}{\sum_{i=0}^{N} \mu_A(x_i)}$$

To implement this in Haskell a we exploit the *uncurry* function of type $(a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$; by partially applying this with the multiplication function $(*)$ we obtain a function which multiplies the two component of a tuple together, mapping this over the representation list a we implement the $x_i \times \mu_A(x_i)$ component. A list of just the degree of memberships $(\mu_A(x_i))$ is obtained trivially by mapping *snd*. Then the *sum* function is used for $\sum$ thus:

$$centroid\ (DF\ a) = sum\ (map\ (uncurry\ (*))\ a)\ /\ sum\ (map\ snd\ a)$$

5.4.2 The ShoeG module with discrete sets

To utilise the newly defined *DF* instance of *FuzzyRep* we shall again use the *shoe_size* fuzzy system. Below we see the consequent fuzzy sets redefined using *singleton* to return *DF* representation sets. Values for the singletons have been chosen such that the single domain value is equal to the domain value which had a degree of membership of one in the *PLF* definition.

small = *singleton* 4 :: *DF*
average = *singleton* 7 :: *DF*
big = *singleton* 10 :: *DF*
very_big = *singleton* 13 :: *DF*

All remaining code for the *shoe_size* function remains the same as given in section 5.3.3 and testing of the module is presented in review Section 5.4.4.

5.4.3 Addition of new operators

We have seen in the previous section how partial application may be used to provide existing Haskell functions (namely product $(*)$ and addition $(+)$) as co-domain operators (for implication and aggregation respectively). Now we demonstrate how two other operators maybe easily defined in Haskell and utilised in a fuzzy system.

1. Firstly we wish to replace the product t-norm with the Łukasiewicz t-norm for our implication operator. It is defined thus:

   $$\star_{\text{luk}}(a, b) = \max\{0, a + b - 1\}$$

   This may be implemented in Haskell as follows:

   $$tLuk\ a\ b = max\ 0\ (a + b - 1)$$

[10] Recall in Section 2.2.1 we defined that any domain value not present in the list implicitly has a zero membership grade.

Module		Height										Work
		1.5	1.55	1.6	1.65	1.7	1.75	1.8	1.85	1.9	1.95	
ShoeG	Orig.	4.67	5.6	6.67	7.0	7.75	9.25	10.0	10.33	11.40	12.33	97,878
ShoeG	Disc.	4.0	4.75	6.25	7.0	7.75	9.25	10.0	10.75	12.25	13.0	27,525

Table 5.3: Testing of *Discrete* module.

2. Secondly we shall replace the addition operator chosen by Meehan and Joy with the bounded sum given by:

$$\oplus_{\text{luk}}(a, b) = \min\{a + b, 1\}$$

With corresponding Haskell implementation of:

$sLuk\ a\ b = min\ (a + b)\ 1$

With the new operators defined it is trivial to adapt the *shoe_size* fuzzy system to use them, the two changes one must make are shown below:

1. The implication definition becomes:

   $(\Rightarrow) = implies\ tLuk$

2. Whilst the aggregation definition becomes:

   $aggregate\ sLuk$

5.4.4 Review of implementation

Due to both a discrete representation and the introduction of the Łukasiewicz operators being demonstrated we review the accomplishments and failings of each one at a time. In conclusion we survey what the implementation have added to the project and what has been learned.

A discrete representation

Accomplishments. In line with previous implementations the *shoe_size* fuzzy system was used for testing. In table 5.4.4 we see the original *PLF* consequent fuzzy sets from Section 5.3.3 compared to the singleton consequent sets defined in Section 5.4.1.

We can see the use of singleton fuzzy sets has the effects of both altering the results slightly and by virtue of the simplified defuzzification process a greatly reduced amount of computation is performed. In this sense the implementation has been a success. However several issues were raised during the implementation and we look at these now:

Failings. Because we chose not to implement the functions *up*, *down* and *top* because they have no natural definition for a discrete set there exists a potential problem. As noted by design the Haskell compiler will not object to the omission of the definition a function from an instance declaration, even if it is present in the class definition. However the Glasgow Haskell Compiler produces warnings as follows.

```
Discrete.lhs:23:3:
    Warning: No explicit method nor default method for 'up'
             In the instance declaration for 'FuzzyRep DF'
```

Were a user of the system to ignore such warnings and utilise an unimplemented function such as *up* to attempt to create a *DF* representation fuzzy set execution would result in an run time error such as:

```
Testing: Discrete.lhs:23:3:
No instance nor default method for class operation FuzzyG.up
```

As given by the Glasgow Haskell compiler or similarly Hugs which yields the similar error message:

```
Program error: undefined member: up
```

Having different combinations of parametric definitions for different representations also raises another issue on top of the possibility of run time errors, it regards the potential requirement to modify the *FuzzyG* module. We touched on this in the implementation where it was noted that because *only* the *DF* representation uses the *singleton* function (which we see from its type *Double* → *DF*) we didn't need to make it part of the *FuzzyRep* class. Were another representation to be developed which had a *singleton* implementation we must now add *singleton* to the minimal definition for the *FuzzyRep* class (so its type becomes *Double* → *FuzzyRep a*). Critically however this would require modifying the *FuzzyG* module; something not possible were the module to be part of a fuzzy logic module set for Haskell (Section 1.1.2).

New co-domain operators

It was shown how the Łukasiewicz t-norm and its dual t-conorm could be easily given as function definitions. Comparing them to the mathematical definitions also given we can see a clear demonstration of how functional languages yield an executable notation very close to the mathematical one.

Next through partial application we were able to replace the existing operators with the two new ones simply by changing two lines. Two approaches were shown:

- By defining a new operator ⇒ and defining it with a partial application of the *implies* function we were able to change the t-norm used for implication without making any alteration to the *shoe_size* function.
- Alternatively we shown how the function can be specified directly within the fuzzy system definition as with the *aggregate* function.

This short implementation also demonstrates the representation abstraction introduced in Section 5.3 clearly; having provided a new set of dual t-norms once they shall work on any representation implemented as an instance of the *FuzzyRep* class.

What has been learned

In this final stage of implementation we have sought to investigate how the *FuzzyG* module and *shoe_size* fuzzy system may be extended. With regard to the introduction of new classes of operators we have seen the approach is very much a success; this is due to the abstraction of representation from co-domain operator.

Unfortunately this success is tarnished by the restriction Haskell imposes that the complete minimal definition for a class, namely *FuzzyRep* must all be given in one file. As such we cannot offer the extensibility of allowing users to add new parametric membership function definitions for multiple representations, whilst maintaining a fixed *FuzzyG* module. The scope for resolving this issue with further work is discussed in Section 6.2.3.

Chapter 6

Evaluation of work

6.1 Summary of chapter

In concluding the project report a full evaluation of the work completed is given. The first part of this shall be a critical review of the implementation as a whole. This shall tie together the three implementation reviews given in Sections 5.2.3, 5.3.4 and 5.4.4; however here we shall not approach each of the three individually; instead we look firstly at the success and limitations of the project then suggest further work. In Section 3.4 we provided two lists of goals; the first listing fuzzy system features and the second listing what properties of the functional paradigm are to be exploited. This first section shall concentrate primarily on the second of these two lists.

Following this evaluation we focus more on the first list of goals, namely how the implementation has been aided or hindered by the functional paradigm. To support this we briefly list the benefits we have seen which are attributed to the used of the functional paradigm. Then in conclusion we compare those areas which were most affected by the choice of the functional paradigm to possible approaches in other paradigms.

6.2 Evaluation of implementation

6.2.1 Successes

Having successfully substituted van den Broek's piecewise representation for the membership function approach used by Meehan and Joy the abstract class of functions for a fuzzy logic representation was derived. We demonstrated how this allowed multiple representations to be used and how an additional representation could be introduced in to the system.

By taking the *shoe_size* function with an implementation published in literature (and thus one against we could test our extensions) as a basis for a fuzzy system we provided a series of results in the three tables 5.2.3, 5.3.4 and 5.4.4. Inspecting these tables we were able to see the *shoe_size* fuzzy system continued to perform in a similar fashion to the one given by Meehan and Joy. Further we observed the effects of using different representation on both the accuracy and the computational complexity of the inferencing process. Specifically on table 5.2.3 we can see the accuracy of the defuzzified shoe size returned by the Meehan and Joy *Shoe* module tends towards the result computed by the geometric approach of the *ShoePLF* module.

The validity of these results is backed up by the successful abstraction of co-domain operators from representation specific operations; which was another goal identified during the discussion chapter. We saw this demonstrated in the final of the three implementation stages where the Łukasiewicz dual t-norm and t-conorm were defined once through the *tLuk* and *sLuk* functions and then shown to work with any representation which implements **class** *Fuzzy*.

Another potential area for extension of the system was that it should be possible to implement a Mamdani fuzzy system which uses two different representations for each of the antecedent (α) and consequent (β) parts. We showed that this was possible through the demonstration of the *shoe_size* fuzzy system using different combinations of representations first in the *ShoeG* module and again in testing a discrete fuzzy set representation.

6.2.2 Limitations

A first limitation noted during implementation of the discrete fuzzy set representation provided by the type *DF* was that both it and the *PLF* type representation had the same underlying type, a list of $(x, \mu_A(x))$ tuples. However they both described very differently behaving fuzzy sets and thus needed to be distinguishable. A solution using algebraic data types was put forwards and utilising them to wrap up the underlying list of tuple type a distinction could be made. In this sense then it does not present a limitation, but we also saw that the use of an algebraic data type led to the extensive use of both pattern matching (in expressions such as $(DF\ a)$) to allow a to be evaluated to the list of tuples and also use of *DF* to construct the type back from a list of tuples. It is certainly fair to say that this lead to code which is less readable than that for the case where an algebraic type is not used such as with the *BCF* and *PLF* representations. It is suggested that perhaps these representation should also be wrapped up in algebraic data types for the purpose of consistency.

A source of difficulty was also identified when we came to introduce the discrete representation to the *FuzzyG* module from the section before it. This stemmed from a lack of agreement between it and the complete minimal definition we had derived from the *BCF* and *PLF* representations, embodied by the *FuzzyRep* class. This disagreement is not an unreasonable situation to have because certain parametric definition simply are not appropriate to certain representations. Two approaches were put forwards to solving this, both bringing their own extra problems:

1. Bypass the *FuzzyRep* class altogether and simply provide a parametric function which returned something of the type of the representation. We saw this approach used in defining the $singleton :: Double \rightarrow DF$ function. This approaches limitation is that we are now unable to implement a *singleton* parametric function for any other representation (if it is to be used at the same time). Attempting to do so would not be allowed because the two functions would have different types.

2. The correct way to deal with the need for ad-hoc polymorphism is by using type classes as was demonstrated with the *up*, *down* and *tri* functions. However the limitations of this approach are twofold:

   (a) It requires the function to be overloaded to be added to the class definition given in the *FuzzyG* module; something not possible were the modules to be released for use in other Haskell projects.

   (b) Once these extra functions (such as *singleton*) become part of the complete minimal definition for the *FuzzyRep* class the Haskell compiler shall expect all representations to implement them, any which do not shall produce warnings.

In either case policing which representation could be used with which parametric definition was shown to be handled as well as could be expected by the compiler. Due to strong typing it was able to detect and produce warnings where an **instance** *FuzzyRep* has undefined parametric function definitions, it is up to the user of the system then to ensure they have not attempted to use them. We could say then that given this limitation the implementation goes some way to limiting the damage it could cause.

6.2.3 Further work

Following on from the limitations discussed in the previous section suggestions are made as to how the system may be developed to address them. Once these have been covered we put forward several other suggestions for work.

The largest obstacle to this projects success was identified as the inability to extend the definition of **class** *FuzzyRep* without modifying the *FuzzyG* module. Two approaches requiring further work are now suggested:

- Firstly one may develop a "mix in" system where by new functions (such as new parametric definitions and new defuzzifiers) are placed in their own class which requires the any **instance** of it to be an instance of the *FuzzyRep* class. In this way it may be possible to maintain a basic tool kit of representations and functions in the *FuzzyG* module whilst allowing them to be used by with our requiring modification of the module.

- A second potential would utilise the techniques put forward in "Data types a la carte" by Wouter Swierstraw [31] where the problem posed is tackled directly.

In Section 5.3.4 we noted a limitation of the approach for discretising continuous fuzzy sets (such as given by the *BCF* representation) where 100 sampling points were always used. Given the data obtained in table 5.2.3 where the number of sampling points was adjusted by altering the implementation for **instance** *FuzzyRep BCF* in the *FuzzyG* module we can see the significant effect this has on both the accuracy and the computational over head of the inferencing. Work would therefore be well merited in identifying a better solution to this problem allowing the discretisation level to be altered without modifying the *FuzzyG* module. Scope for this may well be found in further exploitation of the algebraic data type approach introduced in Section 5.4.1 by making the discretisation level an explicit parameter of the representation.

With a system which allows different fuzzy set representations to be encapsulated there is scope to investigate 2-type fuzzy sets. This would certainly require some modification of the code presented in this report but it is felt the representation abstraction concept does offer potential.

6.3 Suitability of functional paradigm to fuzzy logic

6.3.1 Benefits of functional paradigm

- We have seen how partial application as a benefit of having higher order functions has allowed the development of a fuzzy system where we can utilise complex underlying fuzzy set representations (which allow optimisations when they are manipulated) but obtain a membership function by using the *apply* function.
- To manage these multiple representations and allow them all to be used by one fuzzy system function shown with *shoe_size* whilst minimising the code modification required we exploited ad-hoc polymorphism heavily. In the Haskell language we saw how this polymorphism was obtained by using the type class system and type variables to specify exactly what functions a representation must provide.
- The type class system also allowed us to define define *default* definitions for functions such as *aggregation*. This capitalised on the fact that we could derive the function from other functions with specific **instance** *FuzzyRep* and 'glue' them together using the composition operator $\circ$.
- Having identified that computing the conjunction or disjunction between two fuzzy sets required only the comparison of co-domain values we demonstrated how this allowed an abstraction between representation and the t-(co)norm used. To do this the *junction* function was given as a required function of the *FuzzyRep*. The property of partial application allowed us to use this to great affect in defining the implication operator $\Rightarrow$ in a way which allowed it to work with any representation which supported **instance** *FuzzyRep* and simultaneously allowed the t-norm it used to be easily changed.
- The task of providing usable fuzzy logic modules for Haskell was a limited success. Whilst the limitation of having an entire **class** definition in one continuous block placed by Haskell does make the current suitability of the modules questionable it is easy to how different representation may be brought together as was shown in the final version of *shoe_size* using *DF* type fuzzy sets.

6.3.2 Comparison with other paradigms

During this section the author shall strive for fairness in considering whether we have indeed shown the functional paradigm to be preferential to others; however given that the project's research goal was specifically targeted at the functional paradigm it will not be possible to explore other paradigms as deeply.

An issue not discussed in previous evaluation sections (by virtue of it all being written in Haskell) was an analysis of the actual number of source lines of code which were used. Comparing the implementation given in this report to a similar one the author is confident the number of lines if substantially lower than would be possible in another language[1].

To support this statement we must discuss why the reduction in line count when compared to other languages exists, what the alternatives would be in other paradigms and what benefits a reduction brings. So we firstly we propose two reasons for the reduction:

1. A reduction in the amount of repeated code in a system nets a reduction in the total number of lines. Within this report we have seen several instances where code is defined once and then used in multiple times, for example:
   - The Łukasiewicz t-norm and t-conorm functions *tLuk* and *sLuk* were only defined once, we could then use their definition with all three given representations.
   - Default class functions such as *aggregate* capitalised on the use of type classes to automatically select the correct implementations of the functions which compose them.

[1]This statement assumes an implementation of equal size in terms of features available and the degree to which it could be configured.

Certainly in a non functional programming language code reuse is possible. However the guarantee of referential transparency, that is the guarantee that a function always returns the same output for a given input makes this reuse much safer. We do not have to worry that some global state is adjusted by the functions in a way which makes their use in appropriate for certain representations.

2. The conciseness of the notation used has been demonstrated several times, explicitly in Section 3.3.2 with the definition of a centroid defuzzifier for discrete sets. In cases like this we often see many functions referenced on one line of code and we may rely on associativity (either defaulting to function application binding tightest) or the use of functions such as $\circ$ to control the application of them. In an imperative language we are more likely to see these steps broken down in to a sequence of step by step assignments, each occurring on their own line.

Whilst developing the code given in this report the author primarily used the Hugs interpreter (introduced in Section 2.4.2) this combined with referential transparency guaranteeing there is no global state to affect computation leads to a fast and reliable development cycle. In other languages a series of test cases may have to be developed, due to system state these must be executed in the correct order and further checks must be made that there is no sequence of events which could cause the system to function incorrectly. However using an interpreter one may ensure each function operates as required and then be confident that any functions composed of these shall also work correctly.

Bibliography

[1] Simon Peyton-Jones. A taste of haskell. In *OSCON*, 2007.

[2] J.M. Mendel. *Uncertain Rule-based Fuzzy Logic Systems: Introduction and New Directions.* Prentice Hall PTR, 2001.

[3] D. Dubois and H. Prade. Fuzzy Sets and Systems: Theory and Applications. *New York*, 1980.

[4] G.J. Klir and T.A. Folger. *Fuzzy sets, uncertainty, and information.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1987.

[5] G.J. Klir, U.S. Clair, and B. Yuan. *Fuzzy set theory: foundations and applications.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1997.

[6] A. Kaufmann. *Introduction to the theory of fuzzy subsets.* Academic Press, 1975.

[7] R. Seising. 1965 – "fuzzy sets" apprear – a contribution to the 40th anniversary. *The 2005 IEEE International Conference on Fuzzy Systems*, 2005.

[8] P.E. Johnson. The Genesis and Development of Set Theory. *The Two-Year College Mathematics Journal*, 3(1):55–62, 1972.

[9] LA Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

[10] B. Kosko. *Fuzzy Thinking: The New Science of Fuzzy Logic, Flamingo.* Harper Collins, London, 1993.

[11] L.A. Zadeh. A fuzzy-set-theoretic interpretation of linguistic hedges. *Journal of Cybernetics*, 2(3):4–34, 1972.

[12] JM Garibaldi and RI John. Choosing membership functions of linguistic terms. *Fuzzy Systems, 2003. FUZZ'03. The 12th IEEE International Conference on*, 1, 2003.

[13] L.A. Zadeh. Fuzzy logic and its application to approximate reasoning. *Information Processing*, 74:591–594, 1974.

[14] R. Bellman and M. Giertz. On the Analytic Formalism of the Theory of Fuzzy Sets. *Information Sciences*, 5(149-156):8, 1973.

[15] M. Sugeno. Fuzzy measures and fuzzy integrals: a survey. *Fuzzy Automata and Decision Processes*, pages 89–102, 1977.

[16] EH Mamdani et al. Application of fuzzy algorithms for control of simple dynamic plant. *Proc. IEE*, 121(12):1585–1588, 1974.

[17] P.M. Larsen. Industrial applications of fuzzy logic control. *International Journal of Man-Machine Studies*, 12(1):3–10, 1980.

[18] L.A. Zadeh. *The Concept of a Linguistic Variable and Its Application to Approximate Reasoning.* National Technical Information Service, 1973.

[19] M. EH and S. ASSILIAN. An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller. *International Journal of Human-Computer Studies*, 51(2):135–147, 1999.

[20] PM Van Den Broek. Efficient algorithms for approximate reasoning. *Neural Information Processing, 1999. Proceedings. ICONIP'99. 6th International Conference on*, 1, 1999.

[21] G. MEEHAN and M. JOY. Animated fuzzy logic. *Journal of Functional Programming*, 8(05):503–525, 1998.

[22] PM van den Broek. Fuzzy reasoning with continuous piecewise linear membership functions. *Fuzzy Information Processing Society, 1997. NAFIPS'97. 1997 Annual Meeting of the North American*, pages 371–376, 1997.

[23] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[24] S. Thompson. *The Haskell: The Craft of Functional Programming.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[25] S. Thompson. *Type theory and functional programming.* Addison-Wesley Wokingham, England, 1991.

[26] M.P. Atkinson and R. Morrison. Persistent First Class Procedures are Enough. *Proceedings of the Fourth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 223–240, 1984.

[27] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[28] D.A. Schmidt et al. *The Structure of Typed Programming Languages.* MIT Press, 1994.

[29] The University Court of the University of Glasgow. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[30] OGI and Yale. Hugs. `http://www.haskell.org/hugs/`.

[31] WOUTER SWIERSTRA. Data types a la carte. *Under consideration for publication in J. Functional Programming.*