Fuzzy Functional Programming

Gary Meehan
Department of Computer Science
University of Warwick
E-mail: garym@dcs.warwick.ac.uk

May 1, 1997

Abstract

This report aims to motivate the implementation of fuzzy logic in functional programming in a variety of areas. It includes reasoning using fuzzy logic and fuzzy subsets, and fuzzy expert systems. We show that a functional environment is a natural setting for fuzzy logic, yielding fuzzy programs that are compact, neat, readable and easily adaptable.

1 Introduction

Fuzzy logic [21] is a form of multi-valued logic which finds many applications in expert systems [20], neural nets [4], formal reasoning [14], decision making [14], database enquiries [14] and many other areas. The use of fuzzy logic in such applications not only makes their solutions simpler and more readable but also more efficient and accurate [11, 20].

Fuzzy logic has been applied to many languages — both in extending standard languages such as Prolog [13], Fortran [6], APL [14] and Java [7], and in custom-designed languages such as Fuzzy CLIPS [5], FIL [2, 17], and FLINT [12]. However, no one (to the author's knowledge) has combined fuzziness with a functional language. This report aims to explore the the use of fuzzy logic in functional languages, inspired by the fundamental equivalence of fuzzy subsets and functions (see Section 2).

By using various examples, and describing how they could be written in a functional notation we will show that:

- Using fuzzy logic in a functional context is not inefficient.
- The introduction of fuzzy logic to functional programming preserves the standard advantages of functional programming: readability, compactness and adaptability [1].
- A functional notation introduces greater flexibility. The user is not constrained to one particular fuzzy value set (normally $[0, 1]$) and a handful of standard membership functions.

The functional notation we shall use in this paper is based on Haskell [18].

Section 2 provides a brief overview of fuzzy logic subsets and systems. Sections 3, 4 and 5 explore the use of a functional language to implement solutions of problems in fuzzy logic, subsets and systems respectively. Section 6 contains some concluding remarks. Appendix A provides the original source as well as their functional forms for the three sample fuzzy expert systems used in this report and Appendix B describes some standard fuzzy subsets.

2 A Brief Overview of Fuzzy Logic, Subsets and Systems

For a more detailed examination of fuzzy logic and fuzzy subsets see[9]. A description of fuzzy systems may be found in [8] or [11].

2.1 Fuzzy Logic

In standard (boolean) logic all well-formed expressions evaluate to one of two values in the set $\mathbf{B}$: T or F. In fuzzy logic, we replace the two-valued set by a many-valued set. Typically we use $[0,1]$ where 0 represents absolute falseness, 1 absolute truth and the values in between represent degrees of truthness. This is not the only value set, there are others such as $\{No,\ Maybe,\ Yes\}$ for example.

Given a value set, M, we have the connectives $\wedge_M$, $\vee_M$ and $\neg_M$. These connectives are analogous to the boolean connectives $\wedge$, $\vee$ and $\neg$.

There are various ways in which we can define connectives over M. If $M = [0,1]$ then we have:

$$\begin{aligned} a \wedge_M b &= \min(a,b) \\ a \vee_M b &= \max(a,b) \\ \neg_M a &= 1-a \end{aligned} \tag{1}$$

Or, alternatively:

$$\begin{aligned} a \wedge_M b &= a.b \\ a \vee_M b &= a+b-a.b \\ \neg_M a &= 1-a \end{aligned} \tag{2}$$

When defining connectives we usually require (though it is not essential) that they obey the normal laws of logic: associativity, commutativity, distributivity, de Morgan's laws and the double negative law[1]. For $x, y, z \in M$ we have:

$$\begin{aligned} x \wedge_M (y \wedge_M z) &= (x \wedge_M y) \wedge_M z \\ x \vee_M (y \vee_M z) &= (x \vee_M y) \vee_M z \\ x \wedge_M y &= y \wedge_M x \\ x \vee_M y &= y \vee_M x \\ x \vee_M (y \wedge_M z) &= (x \vee_M y) \wedge_M (x \vee_M z) \\ x \wedge_M (y \vee_M z) &= (x \wedge_M y) \vee_M (x \wedge_M z) \\ \neg_M x \vee_M \neg_M y &= \neg_M (x \wedge_M y) \\ \neg_M x \wedge_M \neg_M y &= \neg_M (x \vee_M y) \\ \neg_M (\neg_M x) &= x \end{aligned}$$

We may also have identities and annihilators for $\wedge_M$ and $\vee_M$ denoted $true_M$ (absolute truthness) and $false_M$ (absolute falsehood) such that for $x \in M$:

$$\begin{aligned} true_M \vee_M x &= x \vee_M true_M &&= true_M \\ true_M \wedge_M x &= x \wedge_M true_M &&= x \\ false_M \vee_M x &= x \vee_M false_M &&= x \\ false_M \wedge_M x &= x \wedge_M false_M &&= false_M \end{aligned} \tag{3}$$

The unit interval with connectives as defined in (1) and (2) satisfy all these laws (see [9] for proof) with *true* taking the value 1 and *false* taking the value 0.

If the equations in (3) hold then we can define an isomorphism $\phi : \{true_M, false_M\} \to \mathbf{B}$ such that:

$$\begin{aligned} \phi(true_M) &= T \\ \phi(false_M) &= F \end{aligned}$$

[1] Idempotence is not required: for instance see eqs. (2).

from which it follows that:

$$\begin{aligned}
\phi(a \wedge_M b) &= \phi(a) \wedge \phi(b) \\
\phi(a \vee_M b) &= \phi(a) \vee \phi(b) \\
\phi(\neg_M a) &= \neg \phi(a)
\end{aligned}$$

where $a, b \in \{true_M, false_M\}$. The proof follows by enumerating all possible choices for a and b. For $\wedge_M$ we have:

$$\begin{aligned}
\phi(true_M \wedge_M true_M) &= \phi(true_M) &= T &= T \wedge T &= \phi(true_M) \wedge \phi(true_M) \\
\phi(true_M \wedge_M false_M) &= \phi(false_M) &= F &= T \wedge F &= \phi(true_M) \wedge \phi(false_M) \\
\phi(false_M \wedge_M true_M) &= \phi(false_M) &= F &= F \wedge T &= \phi(false_M) \wedge \phi(true_M) \\
\phi(false_M \wedge_M false_M) &= \phi(false_M) &= F &= F \wedge F &= \phi(false_M) \wedge \phi(false_M)
\end{aligned}$$

For $\vee_M$ we have:

$$\begin{aligned}
\phi(true_M \vee_M true_M) &= \phi(true_M) &= T &= T \vee T &= \phi(true_M) \vee \phi(true_M) \\
\phi(true_M \vee_M false_M) &= \phi(true_M) &= T &= T \vee F &= \phi(true_M) \vee \phi(false_M) \\
\phi(false_M \vee_M true_M) &= \phi(true_M) &= T &= F \vee T &= \phi(false_M) \vee \phi(true_M) \\
\phi(false_M \vee_M false_M) &= \phi(false_M) &= F &= F \vee F &= \phi(false_M) \vee \phi(false_M)
\end{aligned}$$

And finally for $\neg_M$ we have:

$$\begin{aligned}
\phi(\neg_M true_M) &= \phi(false_M) &= F &= \neg T &= \neg\phi(true_M) \\
\phi(\neg_M false_M) &= \phi(true_M) &= T &= \neg F &= \neg\phi(false_M)
\end{aligned}$$

This shows that we can regard fuzzy logic as an extension of boolean logic. We will drop the subscripts from the connectives and the identities and rely on the context that they are used in to dictate the correct usage.

2.2 Fuzzy Subsets

Consider a subset Y of a set X. For every $x \in X$ either $x \in Y$ or $x \notin Y$. So we can define a characteristic (membership) function $\mu_Y : X \to \{0, 1\}$ such that:

$$\begin{aligned}
\mu_Y(x) &= 1, \textbf{if } x \in Y \\
&= 0, \textbf{otherwise}
\end{aligned}$$

In a fuzzy subset[2] Z of X, instead of each element of X being either in Z or not in it, it is always in Z 'to some degree', that degree being dictated by some fuzzy value set, M say. This value is calculated by evaluating the characteristic function $\mu_Z(x) : X \to M$ for each $x \in X$. Thus a fuzzy subset, Z is a set of pairs which is a subset of $X \times M$ defined as:

$$Z = \{(x, \mu_Z(x)) \mid x \in X\}$$

Note that a fuzzy subset and its characteristic function have the same definition if we take the set-theoretic definition of a function (a set of domain-range pairs). This is an equivalence which we shall exploit later.

Suppose we have the value set $[0, 1]$. If, for some $x \in X$, $\mu_Z(x) = 0$ then x is definitely not in Z. Conversely, if $\mu_Z(x) = 1$ then x definitely is in Z. If $\mu_Z(x) = 0.1$ then we can say that x is in Z 'a bit', and if $\mu_Z(x) = 0.9$ we can say that x is in Z 'a lot'.

[2]Technically, there is no such thing as a fuzzy set, only a fuzzy *sub*set, since before we can start defining fuzzy sets we need some underlying set of possible members, but we shall usually abuse notation and refer to fuzzy sets.

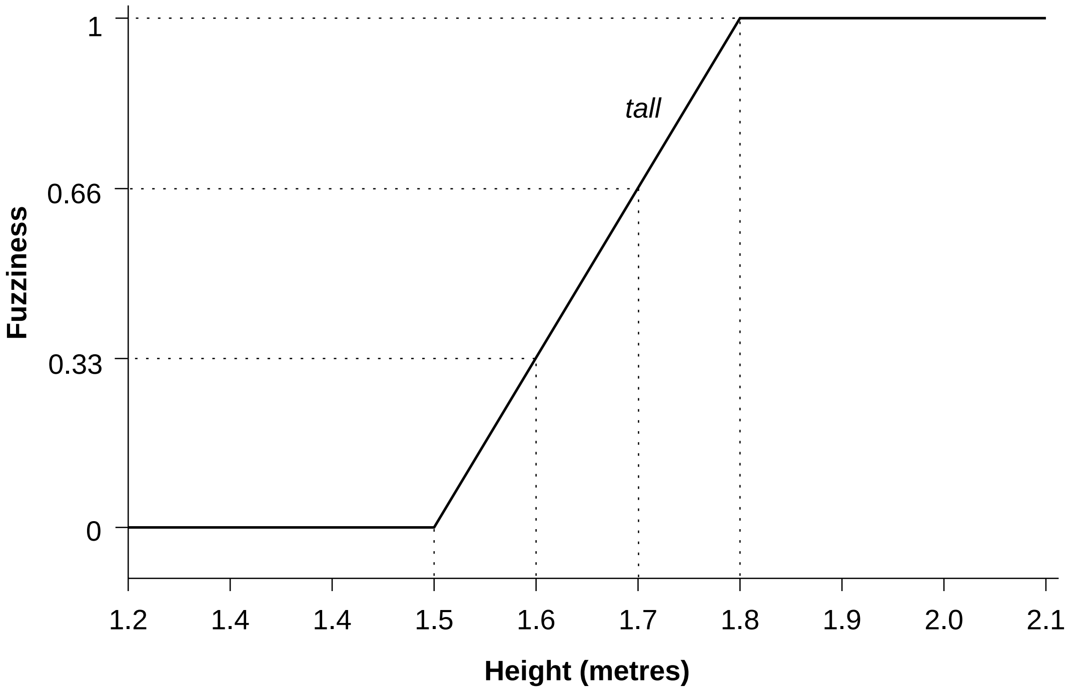

Figure 1: The fuzzy subset *tall*

Thus fuzzy subsets allow us to represent linguistic statements such as 'John is tall a lot' by defining a fuzzy subset *tall* on the set of heights, and for each height assigning a fuzzy value. Fuzzy subsets can be shown graphically by plotting their membership function, for example Figure 1 shows a possible membership function of *tall*. From this we can deduce that $\mu_{tall}(1.5) = 0$, i. e. somebody of 1.5 metres is 'not at all' tall, $\mu_{tall}(1.6) = 0.33$, i. e. somebody of 1.6 metres is tall 'a bit', and so on.

Fuzzy subsets can be combined using the standard operatives such as union. These are defined in the standard way, using the connectives of the underlying fuzzy value set. If we have fuzzy subsets A and B of a set X then we define union, intersection and complement by means of their membership functions for $x \in X$, *viz*:

$$\begin{aligned}
\mu_{A \cup B}(x) &= \mu_A(x) \vee \mu_B(x) \\
\mu_{A \cap B}(x) &= \mu_A(x) \wedge \mu_B(x) \\
\mu_{A^c}(x) &= \neg \mu_A(x)
\end{aligned}$$

2.3 Fuzzy (Expert) Systems

Expert systems are used to model real-world systems in many areas of expertise [16]. Expert Systems consists of sets of *rules* together with an *inference engine* which is used to manage these rules. Rules have an antecedent (an expression in boolean logic) and a consequence, usually assignments to variables and/or manipulation of a facts database.

For instance, suppose we have a shower with two taps, hot and cold, which are used to control the temperature (in °C) and flow (in l/s) of the water [5]. The aim is to get both the temperature and the flow in an acceptable range.

This job can be done by an expert system. Given that the taps take values in the range $[0, 1]$ we might have rules such as the following:

```
(defrule cold_weak
    (outTemp < 36)
    (outFlow < 12)
```

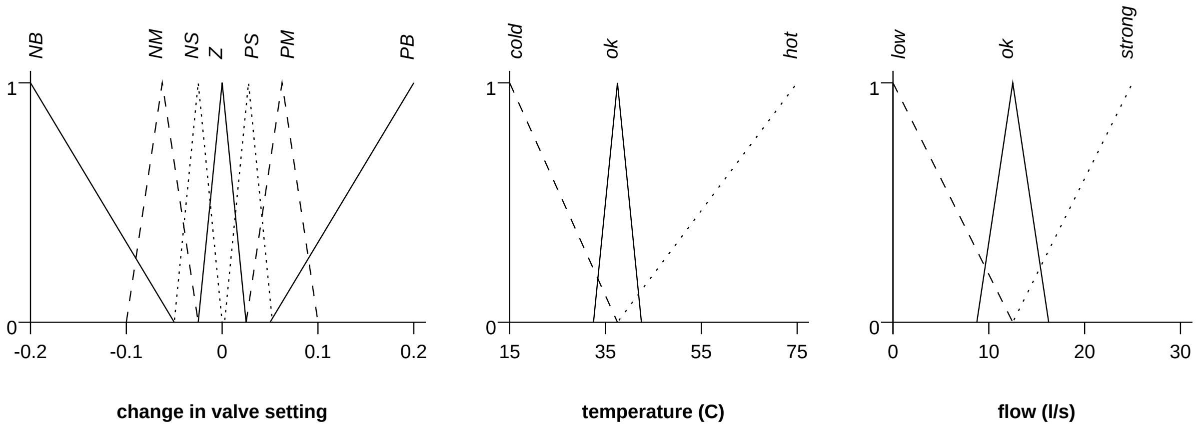

Figure 2: Fuzzy subsets of change, temperature and flow in a shower

```
=>
(assert (change_vh 0.15))
(assert (change_vc 0)))
```

The notation is based on CLIPS [15]. Note that there is an implicit *and* between the two parts of the consequence.

This rule states that when the temperature of the water is below 36°C and the flow is below 12 l/s we increase the hot valve by 0.15 and leave the cold valve alone. The reliance on boolean logic means that rules either fire (i. e. perform the actions of their consequence) or they don't fire. The order in which the rules fire is non-deterministic, though ordering can be specified by the programmer.

If we use fuzzy logic instead, we can have rules which fire to some degree, dictated by the now fuzzy value of the antecedent. We replace the inequalities in the antecedent (e. g. `outTemp < 36`) by fuzzy membership tests (e. g. `outTemp cold`). The assignment of a single value to a variable in the consequence is replaced by the assignment of a fuzzy subset to a variable. These subsets can be seen graphically in Figure 2. This subset will be linearly weighted by the value of its antecedent. For instance, if the antecedent has value a and the consequence is a fuzzy subset with membership function μ, then the weighted consequence will have membership function μ' where $\mu'(x) = a.\mu(x)$.

Fuzzy expert systems can *combine the consequences* of their rules. Suppose we have two rules:

```
(defrule cold_low
    (outTemp cold)
    (outFlow low)
    =>
    (assert (change_vh PM))
    (assert (change_vc Z)))

(defrule OK_low
    (outTemp OK)
    (outFlow low)
    =>
    (assert (change_vh PS))
    (assert (change_vc PS)))
```

where `PB` = Positive Big, `PM` = Positive Medium, `PS` = Positive Small, `Z` = Zero, `NS` = Negative Small, `NB` = Negative Medium and `PB` = Negative Big. The full rule base can be seen in Appendix A.3.

Both the fuzzy subsets `PM` and `PS` are associated with `change_vh`, and similarly for `change_vc`. To resolve this conflict, we combine the consequences (which are weighted fuzzy subsets) into a single consequence, using a combinator such as union or sum. The sum of two fuzzy subsets A and B (which we shall write as $A + B$) has the membership function defined by:

$$\mu_{A+B}(x) = \mu_A(x) + \mu_B(x)$$

The sum method is regarded as the best [11].

A single value, rather than a fuzzy subset, is normally required. This value is obtained by *defuzzifying* the fuzzy subset by taking the subset's centroid or returning one of the elements of the subset with the maximal fuzzy value[3].

One of the most important thing about fuzzy systems is that they work without an underlying mathematical model: the rules are constructed using empirical data. This means we can model complex, non-linear systems using only a few fuzzy rules. Indeed, many systems that cannot be modelled in standard expert systems may easily be modelled in fuzzy systems. It has been shown that any system may be modelled using fuzzy rules [11, 20], though naturally the more complex the system, and the greater the accuracy needed, the more rules may be needed. We have well-defined methods of constructing these fuzzy rules [3, 10].

3 Fuzzy Logic *via* Functional Programming

The notion of translating fuzzy logic into a functional context presents no major problems. All we have is a new type and some functions which manipulate that type. The definition of these functions is a straightforward translation from their corresponding mathematical ones.

3.1 Overloading and Multiple Fuzzy Value Sets

We choose to overload the operator and function names that were defined previously only on the booleans, so that they work over fuzzy values as well. This can be facilitated using Haskell's type classes. [4].

Our overloading of the boolean operators so that they work on fuzzy values as well as booleans means that we aren't restricted to *one* fuzzy value set. For instance, if we have a type class such as the following:

```
class Logic a where
    true, false :: a
    (&&), (||)  :: a -> a -> a
    not         :: a -> a
```

The values `true` and `false` are constant functions which evaluate to absolute truth and absolute falsehood. We can declare that `Bool` is an instance of this class:

```
instance Logic Bool where
    true    = True
    false   = False
    ...
```

filling in the rest of the definition as normal. We can also define a fuzzy value set, based on the unit interval, *viz*:

```
instance Logic Float where
    true    = 1.0
    false   = 0.0
    x && y  = min x y
    x || y  = max x y
    not x   = 1.0 - x
```

[3] Defuzzification also relieves the technical worry that combining the consequences using sum may lead to values which are outside the legal range.

[4] As the Hugs prelude predicts Armageddon if you alter its prelude, it is probably better to use the Haskell-like Gofer interpreter instead.

This is the logic that we shall presume for the rest of this report. For clarity, we define the type synonym `type Unit = Float`[5] to stand for the unit interval.

We aren't restricted to just these two instances. We can define an instance which uses three values, say. For example:

```
data Three  = No | Maybe | Yes

instance Logic Three where
    true    = Yes
    false   = No

    No && x     = No
    Yes && x    = x
    ...
```

3.2 Quantified Model for Learning Disabilities

Even fuzzy logic on its own is useful. As an example. consider the Quantified Model for Learning Disabilities [6] which aims to predict whether a child has learning difficulties. A child is said to have learning difficulties if:

1. The child has a normal achievement potential *AND*
2. Either:
   (a) The child has the behavior characteristics of learning difficulties. *OR*
   (b) The child has the component (i. e. sight, hearing, etc.) deficits within age-related functions

   AND
3. The child has the trait of significantly low achievement.

Each of these criteria has an associated fuzzy value (in the unit interval). In the case of 2a, 2b and 3, this value is the (fuzzy) disjunction of the results of a set of yes/no questions which have an associated weight (a fuzzy value) associated with them. For instance, one of the questions of 2a is:

> 'Difficulty working indicated in a teacher report.' 0.4

If the answer to the question is 'yes' then the weight (in the above case, 0.4) is used in the overall disjunction; if the answer is 'no' the value 0 is used.

Criterion 1 is a little different. The fuzzy value is calculated from the child's IQ (a natural number below 200) and the percentiles (integers between 1 and 100) of the child's results in a quadruple of verbal tests.

This can be translated into the following functional program. Note that this program relies on the order of the answers adhering to that specified in [6].

```
type Percentile = Int
type Answer     = Bool

-- returns the belief to which we believe a child has learning
-- difficulties based on the child's IQ, the percentiles to which the
-- child falls in in 4 verbal tests, and the answers to three lists of
-- yes/no questions
```

[5] Type synonyms aren't allowed in instance definitions in Haskell.

```
learning_disability :: Int -> [Percentile] -> [Answer] -> [Answer] -> [Answer] -> Unit
learning_disability iq verbal_behavs behav_chars problems low_achievement =
    norm_achieve_potential iq verbal_behavs &&
    eval behav_chars    [0.6, 0.4, 0.3, 0.4, 0.6, 0.4, 0.3, 0.7] ||
    eval problems       [0.8, 0.8, 0.8, 0.7, 0.8,  0.8, 0.6, 0.4, 0.3, 0.4] &&
    eval low_achievement [0.6, 0.5, 0.6, 0.4, 0.5, 0.5, 0.6, 0.3, 0.3, 0.2,
                          0.7, 0.5, 0.6, 0.6]

-- Returns the result of the 1st criterion (normal achievement potential)
-- based on the IQ and the percentiles to which the child falls in in 4
-- verbal tests. The fuzzy values associated with the percentiles are
-- obtained by finding their value on a sigmoidal curve.

norm_achieve_potential :: Int -> [Percentile] -> Unit
norm_achieve_potential iq vbs_percs = adequate_intelligence && adequate_verbal_behaviour
    where
    adequate_intelligence
        | iq >= 80             = 0.8
        | 75 <= iq && iq <= 79  = 0.5
        | 50 <= iq && iq <= 74  = 0.1
        | otherwise            = 0.0
    adequate_verbal_behaviour = or (map sigmoidal vbs_percs)
        where
        sigmoidal p = 1 / (1 + exp (0.2 * (p - 25)))

-- Evaluates the value of a criterion by taking the answers to its
-- questions, using these to select either the appropriate weight of
-- false (0) and taking their disjunction

eval [Answer] -> [Unit] -> Unit
eval cs vs = or (zipWith select cs vs)
    where
    select :: Bool -> Unit -> Unit
    select c fv
        | c         = fv
        | otherwise = false
```

Curiously enough, although Horvath presents this as an exercise in fuzzy logic, some of the yes/no questions used make more sense if they were fuzzified. If we consider our example from above concerning the teacher's report on whether or not the child has difficulty working, then the teacher is forced into a strait-jacket and has to either indicate a completely positive or a completely negative response. It would seem to make more sense to let the teacher express the degree to which s/he believes the child to have difficulty working. So, we replace the yes/no questions by letting the teacher dictate to what degree they believe the question to be true. For those questions which *are* strictly yes/no questions, then the teacher can still use the constant functions `true` and `false`.

This increased flexibility actually makes the program simpler. Each boolean answer is replaced by a fuzzy one, which we naturally use to weight (linearly) the value associated with the answer — the values `true` and `false` lead to the same behavior as before.

So, we need to make two changes. The type synonym `Answer` changes to:

```
type Answer = Unit
```

And the definition of `evaluate` changes to:

```
evaluate [Answer] -> [Unit] -> Unit
evaluate cs vs = or (zipWith (*) cs vs)
```

4 Fuzzy Subsets in a Functional Context

As we have already seen in Section 2.1, fuzzy subsets and functions are fundamentally equivalent. So, if we have a type `a` which we want to fuzzify, and some type, `f` say, representing the fuzzy value set, then a fuzzy subset over `a` is simply a function of type `a -> f`, this function is also the membership function of the fuzzy subset. To make it clear we are dealing with a fuzzy subset, we can define the type synonym

```
type FSet a f = a -> f
```

If we consider the *tall* example of Section 2.2, then we can define this functionally as:

```
type Height = Float

tall :: FSet Height Unit
tall = lslope 1.5 1.8
```

where `lslope` takes the value *false* (0) below 1.5, *true* (1) above 1.8 and increases linearly between the two. This is a standard fuzzy subset (see Appendix B) and can be defined by:

```
lslope :: Float -> Float -> FSet Float Unit
lslope a b = \x -> if x < a then false
                   else if x > b then true
                   else (x - a) / (b - a)
```

So, to find out how 'tall' we view a height of 1.65 we simply evaluate `tall 1.65` and yield the answer 0.5.

4.1 Fuzzy Filters and Database Enquiries

Extraction of information from a database usually involves the use of boolean logic which can place artificial constraints on the data. Suppose we had a list of companies and we wanted to find those with high sales [14]. Using a boolean query we would have to use some figure at and above which sales become high, e. g. £1,000,000. But then we have the situation that sales of £1,000,000 are considered high, but sales of £999,999.99 are not considered high.

We could represent 'high' as a fuzzy subset. For example:

```
type Sales = Float -- thousands of pounds

high :: FSet Sales Unit
high = lslope 600 1150
```

Of course, we need some way to exploit this. If we consider a database as a list of records, then a query is just a filter, therefore if we redefine `filter`, *viz*:

```
filter :: (a -> f) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x == false  = filter p xs
    | otherwise     = x:(filter p xs)
```

Then to extract the companies with high sales to some (but not zero) degree, we simply have to apply the function `filter (high . sales)` to our database. The function `sales` extracts the sales information from a company record and has type `Company -> Sales` (we ignore the definition of `Company`).

The above only gives the companies who have some degree of high sales, it is desirable to see how high they are, indicated by the fuzzy value obtained by applying `high . sales` to the company. This can be accomplished by a variation of `filter`:

```
filterv :: (a -> f) -> [a] -> [(a, f)]
filterv _ [] = []
filterv p (x:xs)
    | v == false     = filterv p xs
    | otherwise     = (x, v):(filterv p xs)
    where v = p x
```

This so far is a little simple. Things get more interesting when we pass more complicated expressions to the filter. For example, we may also want to consider the profit margin of our companies. If we want to find all those companies with an acceptable profit margin, we can define the fuzzy subset:

```
type Profit = Float -- Percentage

acceptable :: FSet Profit Unit
acceptable = lslope 12 18
```

and thus we can find all those companies with high sales and acceptable profit by using the fuzzy predicate:

```
high_performer :: Company -> Unit
high_performer co = high (sales co) && acceptable (profit co)
```

with `filter` (here `profit` extracts the profit information from a company record).

We aren't restricted to just excluding all those companies with a fuzzy value of `false`. If we rewrite our function above as:

```
high_performer :: Company -> Bool
high_performer co = (high (sales co) && acceptable (profit co)) >= 0.5
```

Then we only include all those companies which are high performers at and over a degree of 0.5 (and, incidentally, revert back to a boolean predicate in our function[6]).

5 Fuzzy Systems in a Functional Environment

The implementation of a fuzzy expert system in a functional setting may seem at first to be a highly complex problem. An Expert System is large collection of rules with a sophisticated inference engine which manages these rules; fuzzy expert systems have the added complexity of consequence combination and defuzzification. In addition, the order in which the rules fire is largely non-deterministic. How can we hope to implement a largely non-deterministic problem in a deterministic functional language?

We should take note of the following facts:

- Expert systems are *input-oriented*, or *data-driven*[7], that is the design of the system is based on enumerating all the inputs and not only quantifies its output (i. e. works out what its value(s) are) but qualifies it (i. e. works out to what these values belong).

  Functional programs are, conversely, *output-oriented*: they are designed around working out values by evaluating functions. We always know what the values belong to, we are just concerned with working them out (with regard to some input).

[6] This suggests we do not need to rewrite `filter` as a fuzzy function, but simply convert the fuzzy predicate to a boolean, i.e. our fuzzy `filter p` can be written as the boolean `filter p'` where `p' x = p x /= false`, say. However, the rewrite leads to less cumbersome programs and so we prefer it.

[7] Here we are considering forward-chaining expert systems. Backward-chaining systems are goal-oriented *à la* Prolog.

- Fuzzy expert systems are much shorter than standard ones and, as such, don't involve long, complex chains of inferences. This has the effect that the non-determinism of fuzzy expert systems is not as important as it is in standard ones[8].

Bearing these in mind we change the nature of fuzzy expert systems so that they are now means of working out values, rather than of working out what happens when a particular event occurs.

We choose to overload the Haskell conditional `|` even though this is a primitive in Haskell and so couldn't be overloaded in practice without a substantial rewrite of one's Haskell compiler or interpreter.

In a fuzzy context, the conditional evaluates the antecedent of each of its branch and weights the corresponding consequence accordingly. When this has been accomplished, all the consequences are combined using a fuzzy subset function such as union or sum. We keep defuzzification separate and explicit, here *via* the `centroid` function. We presume that there exists an overloaded `if ... then ... else if` which works in a similar manner. We shall not consider the `otherwise` and `else` branches of the conditionals as they have no equivalent in the the world of fuzzy expert systems.

Our change in approach, from input-oriented to output-oriented, means that we no longer need an inference engine to organise the firing of rules, combine consequences and perform defuzzification. These tasks are now taken care of by the evaluation mechanism of the particular compiler/interpreter that we are using. This evaluation mechanism is typically graph reduction [1].

5.1 A Fuzzy Shower Controller

Consider again the shower example (from [5]) which was described in Section 2.3. How would we code a functional implementation? We evaluate the change to the hot- and cold-water taps directly. For instance:

```
change_valves :: (Temp, Flow) -> (Change, Change)
change_valves (temp, flow) = (hot, cold)
    where
    (hot, cold) = (centroid (fst changes), centroid (snd changes))
    changes
        | cold temp && low flow     = (pm, z)
        | cold temp && right flow   = (pm, z)
        | cold temp && strong flow  = (z, nb)
        | ok temp   && low flow     = (ps, ps)
        | ok temp   && strong flow  = (ns, ns)
        | hot temp  && low flow     = (z, pb)
        | hot temp  && right flow   = (nm, z)
        | hot temp  && strong flow  = (nb, z)
```

Note that the fuzzy subset `ok` of flow has been renamed `right` to prevent name clashes. The full code can be seen in Section A.3.

5.2 Conditional Branch Combination

We can also make other minor gains. Consider the following:

```
foo
    | a     = consequence
    | b     = consequence
    |...
```

[8] Non-determinism is a dubious advantage of expert systems, anyway. Anyone who has programmed an expert system will no doubt be aware of the hacks involved in trying to stop the system firing off rules at inconvenient times.

The two separate branches are superfluous and we would like to combine them. How exactly we combine them depends on how the consequences are being combined. If we are using union, we write:

```
foo
    | a || b = consequence
    | ...
```

Whereas, if we were using sum combining, we would have to write:

```
foo
    | a + b = consequence
    | ...
```

For the rest of this report, we shall presume union combining and use `||`.

As an example of this, consider the washing machine example from Appendix A.2. Given the degree and type of dirtiness of a wash, we have to calculate the length of the wash.

We have the original definition using the Aptronix Fuzzy Inference Language (FIL):

```
if dirtiness_of_clothes is Large  and type_of_dirt is Greasy
    then wash_time is VeryLong;
if dirtiness_of_clothes is Medium and type_of_dirt is Greasy
    then wash_time is Long;
if dirtiness_of_clothes is Small  and type_of_dirt is Greasy
    then wash_time is Long;

if dirtiness_of_clothes is Large  and type_of_dirt is Medium
    then wash_time is Long;
if dirtiness_of_clothes is Medium and type_of_dirt is Medium
    then wash_time is Medium;
if dirtiness_of_clothes is Small  and type_of_dirt is Medium
    then wash_time is Medium;

if dirtiness_of_clothes is Large  and type_of_dirt is NotGreasy
    then wash_time is Medium;
if dirtiness_of_clothes is Medium and type_of_dirt is NotGreasy
    then wash_time is Short;
if dirtiness_of_clothes is Small  and type_of_dirt is NotGreasy
    then wash_time is VeryShort
```

The definitions of the various fuzzy subsets can be found in Appendix A.2. This can be written functionally as:

```
wash_time :: Degree -> Degree -> WashTime
wash_time greasiness dirtiness= minimum (maxima wt)
    where
    wt
        | high greasiness && high dirtiness                        = very_long
        | high greasiness && (normal dirtiness || low dirtiness)   = long
        | normal greasiness && high dirtiness                      = long
        | normal greasiness && (normal dirtiness || low dirtiness) = medium
        | low greasiness && high dirtiness                         = medium
        | low greasiness && normal dirtiness                       = short
        | low greasiness && low dirtiness                          = very_short
```

presuming that the distributive laws hold for for `&&` and `||`. Although we still have repeated consequences (two of `long` and of `medium`) the antecedents leading to them have nothing in common, and nothing is gained by combining them into a single branch.

It should be pointed out that we cannot nest conditionals unless `&&` and multiplication (which we use to do the weighting of the consequences) are equal[9]. This means that in general

[9]This is true in the case of the booleans or on the unit range where `&&` is defined as multiplication directly.

```
if a then (if b then c)
```

is not equivalent to

```
if a && b then c
```

5.3 Code Reduction

There are further steps that we can employ to reduce the amount of code needed. Consider the camera auto-focusing example in Appendix A.1 the original of which uses Aptronix's FIL. Here we have to automatically focus a camera, based on the distance to the object that we are focusing on [2].

Because the object(s) that are being photographed may not be at the centre of the image, we base the focusing calculations on three distances. These are the distances to the nearest objects (the focal lengths) on the left, centre and right of the view finder. The fuzzy system works out which one of these is the most plausible focal length.

The input distances each have three fuzzy subsets associated with them — `Far`, `Medium` and `Near`. The definitions of these fuzzy subsets are naturally identical for each of the inputs. Since fuzzy subsets aren't naturally tied to a variable, we can reuse the definitions. So, instead of having

```
invar Left "meter" : 1 () 100 [
        Far      (@10, 0,  @40,  1,  @100, 1),
        Medium   (@1,  0,  @10,  1,  @40,  0),
        Near     (@1,  1,  @10,  0)];
```

repeated for for the three inputs, we simply define functions:

```
type Distance     = Float -- distance in metres

far, medium, near :: FSet Distance
near   = rtri 1 10
medium = atri 1 10 40
far    = ltrap 10 40 100
```

which are applicable for any distance, not just a specific left, right or centre one. A similar reuse of code can be done with the output variables.

There is more scope for redundancy removal in the system. The rules which assign the plausibilities yield identical functions for calculating the plausibilities for the left and right portions of the view-screen. We replace these two functions with a general one. See Appendix A.1.

6 Conclusion

We have explored the use of fuzzy logic in functional programming. The natural equivalence between fuzzy subsets and their membership functions motivates our idea to use a single function to model them both.

We have shown how a functional language can be extended so that it provides facilities for the use of fuzzy logic and fuzzy subsets, achieved by overloading pre-existing operators and functions, and introducing new ones.

We have also shown how, by a change in approach from input-oriented (data driven) to output-driven (evaluation), we can implement fuzzy expert systems in a functional language. This new approach led us to consider how we could make efficiency gains by exploiting the various laws of fuzzy logic (associativity, distributivity, commutativity) to rewrite our functions so that they become shorter and simpler.

Work is currently being done on a practical implementation of the ideas presented in this report. The language of choice is Gofer rather than Haskell, as the former has the advantages of multiple-parameter type classes and a user-rewritable prelude which are necessary to facilitate the heavy use of overloading that we require.

A Sample Fuzzy Systems

A.1 Camera Auto-focusing

The original source uses Aptronix's FIL to calculate the plausibilities for each of the three possible focal lengths.

```
$               This information is provided by the
$                       Aptronix FuzzyNet
$                 http://www.aptronix.com/fuzzynet
$                  email fuzzynet@aptronix.com
$               Aptronix may also be reached by phone
$                       at 408-261-1898 or
$                       FAX at 408-490-2729
$
$
$ FILENAME:     camera/af1.fil
$ DATE:         29/07/1992
$ UPDATE:       06/08/1992

$ Three inputs, three outputs, decision making
$ for Automatic Focusing System
$ INPUT(S):     Left(Distance), Center(Distance), Right(Distance)
$ OUTPUT(S):    Plau(sibility)_of_Left, Plau(sibility)_of_Center,
$               Plau(sibility)_of_Right

$ FIU HEADER

fiu tvfi (min max) *8;

$ DEFINITION OF INPUT VARIABLE(S)

invar Left "meter" : 1 () 100 [
        Far      (@10, 0,  @40,  1,  @100, 1),
        Medium   (@1,  0,  @10,  1,  @40,  0),
        Near     (@1,  1,  @10,  0)
        ];

invar Center "meter" : 1 () 100 [
        Far      (@10, 0,  @40,  1,  @100, 1),
        Medium   (@1,  0,  @10,  1,  @40,  0),
        Near     (@1,  1,  @10,  0)
        ];

invar Right "meter" : 1 () 100 [
        Far      (@10, 0,  @40,  1,  @100, 1),
        Medium   (@1,  0,  @10,  1,  @40,  0),
        Near     (@1,  1,  @10,  0)
        ];

$ DEFINITION OF OUTPUT VARIABLE(S)

outvar Plau_of_Left "degree" : 0 () 1 * (
        VeryHigh = 1.0,
        High     = 0.8,
        Medium   = 0.5,
        Low      = 0.3
    );
```

```
outvar Plau_of_Center "degree" : 0 () 1 * (
        VeryHigh = 1.0,
        High     = 0.8,
        Medium   = 0.5,
        Low      = 0.3
    );

outvar Plau_of_Right "degree" : 0 () 1 * (
        VeryHigh = 1.0,
        High     = 0.8,
        Medium   = 0.5,
        Low      = 0.3
    );

$ RULES

if Left is Near then Plau_of_Left is Medium;
if Center is Near then Plau_of_Center is Medium;
if Right is Near then Plau_of_Right is Medium;

if Left is Near and Center is Near and Right is Near then Plau_of_Center is High;
if Left is Near and Center is Near then Plau_of_Left is Low;
if Right is Near and Center is Near then Plau_of_Right is Low;

if Left is Medium then Plau_of_Left is High;
if Center is Medium then Plau_of_Center is High;
if Right is Medium then Plau_of_Right is High;

if Left is Medium and Center is Medium and Right is Medium then Plau_of_Center is VeryHigh;
if Left is Medium and Center is Medium then Plau_of_Left is Low;
if Right is Medium and Center is Medium then Plau_of_Right is Low;

if Left is Far then Plau_of_Left is Low;
if Center is Far then Plau_of_Center is Low;
if Right is Far then Plau_of_Right is Low;
if Left is Far and Center is Far and Right is Far then Plau_of_Center is High;

if Left  is Medium and Center is Far then Plau_of_Center is Low;
if Right is Medium and Center is Far then Plau_of_Center is Low

end
```

The functional source, with many of the redundancies removed. Although not given in the original source, we have included the final part of the specification in our functional solution which uses normal logic to determine the focal length

```
type Distance     = Float -- [1, 100] distance in metres
type Plausibility = Float -- {0.3, 0.5, 0.8, 1.0}

far, medium, near :: FSet Distance
near   = rtri 1 10
medium = atri 1 10 40
far    = ltrap 10 40 100

low, average, high, very_high :: FSet Plausibility
low       = spike 0.3
average   = spike 0.5
high      = spike 0.8
very_high = spike 1.0

-- edge (e) is either from the left or right
edge_plausibility :: Distance -> Distance -> Plausibility
```

```
edge_plausibility e c = minimum (maxima lp)
    where
    lp
        | near e                    = average
        | medium e                  = high
        | far e                     = low
        | near e && near c          = low
        | medium e && medium c      = low

centre_plausibility :: Distance -> Distance -> Distance -> Plausibility
centre_plausibility l c r = minimum (maxima cp)
    where
    cp
        | near c                            = average
        | medium c                          = high
        | far c                             = low
        | near l && near c && near r        = high
        | medium l && medium c && medium r  = very_high
        | far l && far c && far r           = high
        | (medium r || medium l) && far c   = low

-- not the most efficient w.r.t # comparisons, but it's understandable
focal_length :: Distance -> Distance -> Distance -> Distance
focal_length l c r
    | cp >= lp && cp >= rp  = c
    | lp > cp && lp > rp    = l
    | rp > cp && rp > lp    = r
    where
    lp = edge_plausibility l c
    rp = edge_plausibility r c
    cp = centre_plausibility l r c
```

A.2 Washing Machine

Fuzzy controller to determine the wash time of a load of clothes based on the dirtiness of the clothes and the type of dirt [17]. The two input parameters are split into three triangular fuzzy sets; the output parameter is split into five *crisp* sets (or fuzzy subsets which are just spikes). Defuzzification is done by the min-maxima method, as the output has to be set to one of the times indicated by the output sets.

The original source uses Aptronix's FIL:

```
$               This information is provided by the
$                       Aptronix FuzzyNet
$                 http://www.aptronix.com/fuzzynet
$                  email fuzzynet@aptronix.com
$               Aptronix may also be reached by phone
$                       at 408-261-1898 or
$                       FAX at 408-490-2729
$
$
$ FILENAME:     washmach\wash1.fil
$ DATE:         July 23, 1992
$ UPDATE:       July 29, 1992
$ AUTHOR:       W. Zhang, Aptronix, Inc.
$ REFERENCE:    P94-98, BK(J), M.Nagamachi, 1991, KAIBUNDO

$ CONTROLLER for Washing Machine: Two inputs, one output, open-loop control
$ INPUT(S):     dirtness_of_clothes, type_of_dirt
$ OUTPUT(S):    wash_time

$ FIU HEADER
```

```
fiu tvfi (min max) *8;

$ DEFINITION OF INPUT VARIABLE(S)

invar dirtness_of_clothes "degree" : 0 () 100 [
      Large  (@50, 0,  @100,  1),
      Medium (@0,  0,  @50,   1,  @100,  0),
      Small  (@0,  1,  @50,   0)
    ];

invar type_of_dirt "degree" : 0 () 100 [
      Greasy    (@50,  0,  @100,  1),
      Medium    (@0,   0,  @50,   1,  @100,  0),
      NotGreasy (@0,   1,  @50,   0)
    ];

$ DEFINITION OF OUTPUT VARIABLE(S)

outvar wash_time "minute" : 0 () 60 * (
        VeryLong  = 60,
        Long      = 40,
        Medium    = 20,
        Short     = 12,
        VeryShort = 8
    );

$ RULES

if dirtness_of_clothes is Large  and type_of_dirt is Greasy
    then wash_time is VeryLong;
if dirtness_of_clothes is Medium and type_of_dirt is Greasy
    then wash_time is Long;
if dirtness_of_clothes is Small  and type_of_dirt is Greasy
    then wash_time is Long;

if dirtness_of_clothes is Large  and type_of_dirt is Medium
    then wash_time is Long;
if dirtness_of_clothes is Medium and type_of_dirt is Medium
    then wash_time is Medium;
if dirtness_of_clothes is Small  and type_of_dirt is Medium
    then wash_time is Medium;

if dirtness_of_clothes is Large  and type_of_dirt is NotGreasy
    then wash_time is Medium;
if dirtness_of_clothes is Medium and type_of_dirt is NotGreasy
    then wash_time is Short;
if dirtness_of_clothes is Small  and type_of_dirt is NotGreasy
    then wash_time is VeryShort
end
```

The conversion to a functional notation allows us to remove a few redundancies. The definitions of the fuzzy subsets of the degrees of dirtiness and greasiness in the original source are exactly the same so we combine them into a single set of definitions. We also combine two sets of rule pairs into two single rules using the distributive laws.

```
type Degree    = Float -- [0, 100] percentage
type WashTime  = Int   -- {8, 12, 20, 40, 60} minutes

low, normal, high :: FSet Degree Unit
low    = rtri  0  50
normal = tri   0 100
high   = ltri 50 100
```

```
very_short, short, medium, long, very_long :: FSet WashTime Unit
very_short = spike 8
short      = spike 12
medium     = spike 20
long       = spike 40
very_long  = spike 60

wash_time :: Degree -> Degree -> WashTime
wash_time greasiness dirtiness= minimum (maxima wt)
    where
    wt
        | high greasiness && high dirtiness                         = very_long
        | high greasiness && (normal dirtiness || low dirtiness)    = long
        | normal greasiness && high dirtiness                       = long
        | normal greasiness && (normal dirtiness || low dirtiness)  = medium
        | low greasiness && high dirtiness                          = medium
        | low greasiness && normal dirtiness                        = short
        | low greasiness && low dirtiness                           = very_short
```

A.3 Shower Control

The full text of the original Fuzzy CLIPS program is unavailable. However, the rule base is as follows:

```
(defrule cold_low
  (outTemp cold)
  (outFlow low)
  =>
  (assert (change_vh PM))
  (assert (change_vc Z)))

(defrule cold_OK
  (outTemp cold)
  (outFlow OK)
  =>
  (assert (change_vh PM))
  (assert (change_vc Z)))

(defrule cold_strong
  (outTemp cold)
  (outFlow strong)
  =>
  (assert (change_vh Z))
  (assert (change_vc NB)))

(defrule OK_low
  (outTemp OK)
  (outFlow low)
  =>
  (assert (change_vh PS))
  (assert (change_vc PS)))

(defrule OK_strong
  (outTemp OK)
  (outFlow strong)
  =>
  (assert (change_vh NS))
  (assert (change_vc NS)))

(defrule hot_low
  (outTemp hot)
  (outFlow low)
```

```
  =>
  (assert (change_vh Z))
  (assert (change_vc PB)))

(defrule hot_OK
  (outTemp hot)
  (outFlow OK)
  =>
  (assert (change_vh NM))
  (assert (change_vc Z)))

(defrule hot_strong
  (outTemp hot)
  (outFlow strong)
  =>
  (assert (change_vh NB))
  (assert (change_vc Z))
```

The functional source is as follows. Note that the fuzzy subsets involved do *not* come from the original source but were developed by trial and error.

```
-- new types

type Temp = Float
type Flow = Float
type Change = Float

-- fuzzy subset definitions

cold, ok, hot :: FSet Temp Unit
cold    = rtri 15.0 36.0
ok      = tri  32.0 40.0
hot     = ltri 36.0 75.0

low, right, strong :: FSet Flow Unit
low     = rtri  0.0 12.0
right   = tri   9.0 15.0
strong  = ltri 12.0 25.0

nb, nm, ns, z, ps, pm, pb :: FSet Change Unit
nb  = rtri (-0.2)  (-0.05)
nm  = tri  (-0.1)  (-0.025)
ns  = tri  (-0.05)    0.0
z   = tri (-0.025)  0.025
ps  = tri    0.0     0.05
pm  = tri    0.025   0.1
pb  = ltri   0.05     0.2

-- calculate the change in the hot- and cold-water taps needed
-- given a temperature and flow

change_valves :: (Temp, Flow) -> (Change, Change)
change_valves (temp, flow) = (hot, cold)
    where
    (hot, cold) = (centroid (fst changes), centroid (snd changes))
    changes
        | cold temp && low flow     = (pm, z)
        | cold temp && right flow   = (pm, z)
        | cold temp && strong flow  = (z, nb)
        | ok temp   && low flow     = (ps, ps)
        | ok temp   && strong flow  = (ns, ns)
        | hot temp  && low flow     = (z, pb)
        | hot temp  && right flow   = (nm, z)
        | hot temp  && strong flow  = (nb, z)
```

B Standard Fuzzy Subsets Distributions

The standard fuzzy subset definitions for a base set of real numbers and a unit interval fuzzy value set are given in Figure 3. Not that all bar `lslope` and `rslope` are variants of `trap` — e. g. `ltri a b` = `trap a b b b` — and are given purely for the sake of convenience and readability.

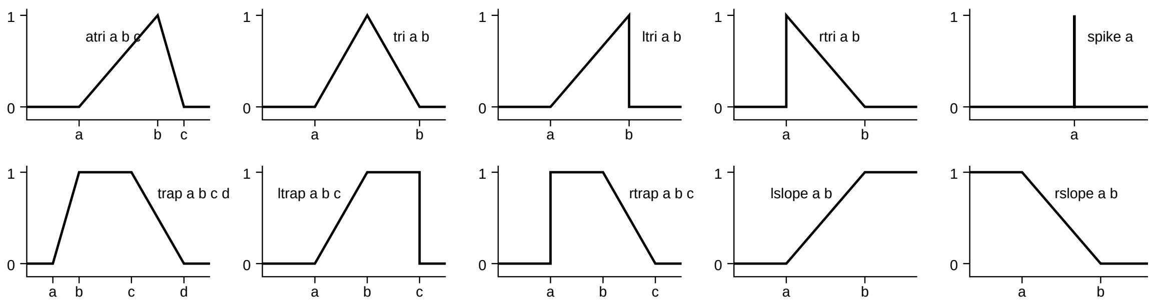

Figure 3: Standard Fuzzy Subset Distributions

References

[1] Richard Bird and Philip Wadler. *An Introduction to Functional Programming.* Prentice Hall, 1988.

[2] Aptronix FuzzyNet. *Focusing System.* WWW: http://www.aptronix.com/fuzzynet /applnote/focusing.htm

[3] Julia A. Dickerson and Bart Kosko. *Fuzzy Function Approximation with Ellipsoid Rules.* IEEE Transactions on Systems, Man and Cybernetics, Vol. 26, No. 4, pp 542–560, August 1996.

[4] Patrik Eklund and Frank Kwalonn. *Neural Fuzzy Logic Programming.* IEEE Transactions on Neural Networks, Vol. 3, No. 5, pp 815–818, September 1992.

[5] NRC-CNC Institute for Information Technology. *Fuzzy CLIPS.* WWW: http://ai.iit.nrc.ca/fuzzy/fuzzy.html.

[6] J. M. Horvath. *A Fuzzy Set Model of Learning Disability.* Fuzzy Sets in Psychology, North Holland, pp 345–381, 1988.

[7] Aptronix Ltd. *Fuzzy Java.* WWW: http://www.aptronix.com/fuzzynet/applnote/java.htm

[8] Mark Kantrowitz, Erik Horstkotte, and Cliff Joslyn. *FAQ: Fuzzy Logic and Fuzzy Expert Systems.* WWW: http://www.cs.cmu.edu/Groups/AI/html/faqs/ai/fuzzy/part1/faq.html.

[9] Arnold Kaufmann. *Introduction to the Theory of Fuzzy Subsets, Volume 1.* Academic Press, 1975.

[10] László T. Kóczy and Kaoru Hirotu. *Size Reduction by Interpolation in Fuzzy Rule Bases.* IEEE Transactions on Systems, Man and Cybernetics, Vol. 27, No. 1, pp 14–25, 1997.

[11] Bart Kosko. *Fuzzy Thinking.* Flamingo, 1994.

[12] Logic Programming Associates Ltd. *FLINT Toolkit.* WWW: http://www.lpa.co.uk/fln.html

[13] T. P. Martin, J.F. Baldwin and B.W. Pilsworth. *The Implementation of FProlog — A Fuzzy Prolog Interpreter.* Fuzzy Sets and Systems, Vol 23, pp 119-129, 1987.

[14] C. V. Negoita. *Expert Systems and Fuzzy Systems.* The Benjamin/Cummings Publishing Company, 1985.

[15] Gary Riley. *CLIPS: A Tool for Building Expert Systems.* WWW: http://www.jsc.nasa.gov/ clips/CLIPS.html

[16] Stuart Russell and Peter Norvig. *Artificial Intelligence — A Modern Approach.* Prentice Hall, 1995.

[17] Aptronix FuzzyNet. *Focusing System.* WWW: http://www.aptronix.com/fuzzynet/ applnote/wash.htm

[18] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison Wesley, 1996.

[19] R. Tong and P. Bonisonne. *A Linguistic Approach to Decision Making with Fuzzy Sets.* IEEE Transactions on Systems, Man and Cybernetics, SMC-10, pp 716–723, 1980.

[20] Li-Win Wang. *Adaptive Fuzzy Systems and Control — Design and Stability Analysis.* Prentice Hall, 1994.

[21] L. A. Zadeh. *Outline of a New Approach to the Analysis of Complex Systems and Decision Processes.* IEEE Transactions on Systems, Man and Cybernetics, Vol. 3, pp 28–44, 1973.