J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press

Animated Fuzzy Logic

Gary Meehan and Mike Joy
Department of Computer Science
University of Warwick
Coventry
UK, CV4 7AL
E-mail: {*Gary.Meehan, M.S.Joy*}*@dcs.warwick.ac.uk*

Abstract

In this paper we aim to give an introduction to fuzzy logic using the language Haskell to implement our solutions. We shall see how the high-level, declarative nature of a functional language allows us to implement easily and efficiently solutions to problems using fuzzy logic and, in particular, how the presence of functions as first-class values allows us to model the key concept of the fuzzy subset in a natural way.

1 Introduction

Fuzzy logic, developed by Lotfi Zadeh (Zadeh, 1965; Zadeh, 1973), is a form of multi-valued logic which has its grounds in Łukasiewicz's work on such logics (Łukasiewicz, 1967a; Łukasiewicz, 1967b). It finds many applications in expert systems (in particular control problems) (Cox, 1994; Mamdani & Assilian, 1975; Ross, 1995; Wang, 1994), neural nets (Eklund & Kwalonn, 1992), formal reasoning (Negoita, 1985; Tanaka, 1997), decision making (Cox, 1994; Negoita, 1985; Zimmermann, 1991), database enquiries (Negoita, 1985) and many other areas. The use of fuzzy logic in such applications not only makes their solutions simpler and more readable but can also make them more efficient, stable and accurate (see, for example, Chapter 2 of (Wang, 1994), or Chapter 3 of (Yan *et al.*, 1994)).

Fuzzy logic has been applied to many languages — both in extending standard languages such as Prolog (Martin *et al.*, 1987), Fortran (Horvath, 1988), APL (Negoita, 1985) and Java (Aptronix Ltd., 1996), and in custom-designed languages such as Fuzzy CLIPS (for Information Technology, 1996), FIL (Aptronix Ltd., 1992a; Aptronix Ltd., 1992b), and FLINT (Ltd., 1997). However, no one, to the authors' knowledge, has combined fuzziness with a functional language.

In this paper we aim to give an introduction to fuzzy logic using the language Haskell (Peterson & Hammond, 1997) to implement our solutions. We shall see how the high-level, declarative nature of a functional language allows us to implement easily and efficiently solutions to problems using fuzzy logic and, in particular, how the presence of functions as first-class values allows us to model the key concept of a *fuzzy subset* (see Section 3) in a natural way.

This paper is arranged as follows. Section 2 introduces the logic part of fuzzy logic

(the term 'fuzzy logic' is used to describe both the actual logic and the whole concept of fuzzy theory). Section 3 introduces fuzzy subsets and some of their applications. Section 4 introduces fuzzy systems and gives several examples. Section 5 concludes.

Throughout the paper we shall give examples of using the programs we develop using the Haskell interpreter Hugs (Thompson, 1996). The programs in question can be downloaded off the WWW from:

http://www.dcs.warwick.ac.uk/people/research/Gary.Meehan/funcprog/research.html

Hugs is available from:

http://haskell.systemsz.cs.yale.edu/hugs/

2 Fuzzy Logic

In fuzzy logic, the two-valued truth set of boolean logic is replaced by a multi-valued one, usually the unit interval $[0, 1]$. Truth sets taking values in this range are said to be *normalised*. In this set, 0 represents absolute falsehood and 1 absolute truth, with the values in between representing increasing degrees of truthness from 0 to 1. So we can say that 0.9 is 'nearly true', 0.5 is 'as true as it is false' and 0.05 is 'very nearly false'. The nearer a value is to 0 or 1 the *crisper* it is; the nearer it is to 0.5 (the middle value of the range) the *fuzzier* it is.

The standard connectives of boolean logic — $\wedge$, $\vee$ and $\neg$ — are adapted so that they work with the fuzzy truth set. There are many ways in which this can be done, but whatever definition we choose we expect the following to hold (Fodor & Roubens, 1994; Zimmermann, 1991):

1. $\wedge$ and $\vee$ should be associative and commutative.
2. $\wedge$ and $\vee$ should be monotonic. That is, if $a, b, c \in [0, 1]$ and $a \leq b$ then $a \wedge c \leq b \wedge c$ and similarly for $\vee$.
3. 1 and 0 are the identities of $\wedge$ and $\vee$ respectively. From this and monotonicity we deduce that 1 and 0 are annihilators of $\vee$ and $\wedge$ respectively.
4. $\neg$ should be anti-monotonic. That is if $a, b \in [0, 1]$ and $a \leq b$ then $\neg b \leq \neg a$. Normally this should be strict monotonicity, that is if $a < b$ then $\neg b < \neg a$.
5. $\neg$ should be its own inverse, that is if $a \in [0, 1]$ then $\neg\neg a = a$.
6. If we restrict the truth set to just 0 and 1, then our logic should behave *exactly* as boolean logic.

Definitions of $\vee$ and $\wedge$ that satisfy the above are also known as *t-norms* and *t-conorms* (or *s-norms*) respectively.

We would also expect the connectives to be continuous and to satisfy DeMorgan's laws. Two definitions which do so, taking values in the set $[0, 1]$, and which are probably the most common are Zadeh's original definition (Zadeh, 1965; Zadeh, 1973) using minimum and maximum operators:

$$\begin{aligned} x \wedge y &= \min(x, y) \\ x \vee y &= \max(x, y) \\ \neg x &= 1 - x \end{aligned}$$

and an alternative using sum and product definitions:

$$x \wedge y = xy$$
$$x \vee y = x + y - xy$$
$$\neg x = 1 - x$$

Note that $p \wedge \neg p = 0 \iff p \in \{0, 1\}$ in both these and most other definitions of fuzzy logic. For instance, $0.3 \wedge \neg 0.3 = 0.3 \wedge 0.7 = 0.3$ using Zadeh's definition, and 0.21 if we use the product definition of $\wedge$. Of course, this is only an elementary introduction to fuzzy logic, and we have not mention more esoteric connectives such as *averaging operators*. For more information we refer the reader to (Kaufmann, 1975), (Zimmermann, 1991) and (Fodor & Roubens, 1994). From now on we shall presume that all fuzzy truth values lie in $[0, 1]$.

We shall now set about implementing these ideas in Haskell. We shall place all our definitions in a module called `Fuzzy` which will redefine some of the functions defined in the Haskell prelude. This is done by *shadowing* the previous definitions (see Section 5.3.2 of the Haskell report (Peterson & Hammond, 1997)). Thus the `Fuzzy` module and any module which wishes to import it should contain the declaration:

```
import Prelude hiding ((&&), (||), not, and, or, any, all)
```

This forces an explicit import of the prelude (which is normally implicitly imported), but hides the functions which we want to redefine. An example of the importing procedure can be seen Section 3.5.

Fuzzy truth values are represented using the Haskell type `Double`. The connectives are implemented by overloading the operators `&&`, `||`, etc. so that they work on fuzzy values as well as boolean ones. This is done by shadowing the connectives (see above) and placing the connectives in a class (Hall *et al.*, 1996; Jones, 1995; Peyton Jones *et al.*, 1997):

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

The functions `and`, `or`, etc. are then also overloaded so that they now operate on instances of the `Logic` class, rather than just the `Bool` type as before:

```
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
```

We can then declare instances of this class — `Bool` is declared in the obvious way (with `true` = `True`, etc.); for fuzzy truth values (values of type `Double`) we have:

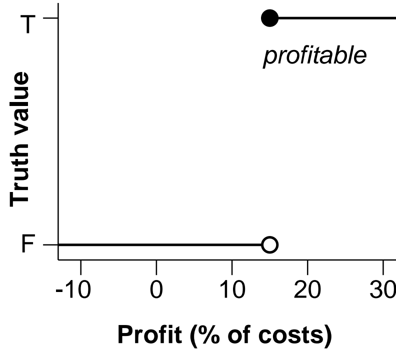

Fig. 1. Crisp definition of Profit.

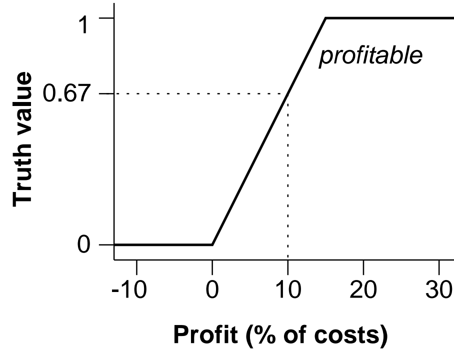

Fig. 2. Fuzzy definition of Profit.

```
instance Logic Double where
    true        = 1
    false       = 0
    (&&)        = min
    (||)        = max
    not x       = 1 - x
```

Note that as with the `Bool` case, `true` is the identity of `&&` and `false` is the identity of `||` (provided we stick with values in $[0, 1]$, of course). So, for example, $0.5 \wedge (0.3 \vee \neg 0.8)$ can be evaluated in Hugs as:

```
Fuzzy> 0.5 && (0.3 || not 0.8) :: Double
0.3
```

where '`Fuzzy>`' is the Hugs prompt. The explicit typing is necessary to resolve the overloading.

3 Fuzzy Subsets

Given a set A and a subset of it, B say, we can define a characteristic (membership) function $\mu_B : A \to \{0, 1\}$ defined such that:

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

This characteristic function determines which elements of A are in B and which are not. Now suppose we replace the two-valued range of μ_B with the unit interval, just as we replaced the boolean truth set with this interval. Then membership of the subset B of A is no longer an absolute but rather something which takes varying degrees of truthness. For $x \in A$, the closer $\mu_B(x)$ is to 1, the more we can regard x as belonging to B, with $\mu_B(x) = 1$ holding if x definitely is in B. Conversely, the closer $\mu_B(x)$ is to 0, the more we can regard x as *not* belonging to B. The subset B is no longer a crisp set but a *fuzzy* one.

A fuzzy subset B of a set A is a set of pairs with each element of A associated with

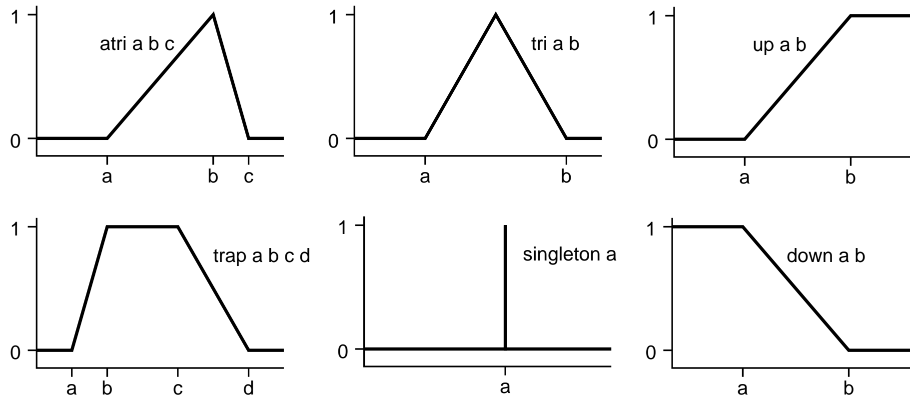

Fig. 3. Standard fuzzy subset distributions

the degree to which it belongs to B (determined by μ_B). Formally, $B \subset A \times [0,1]$ where $B = \{\langle x, \mu_B(x)\rangle \mid x \in A\}$

Given the set-theoretic definition of a function, that is a set of domain-range pairs, we note that *the definition of B and its characteristic function are identical.* This is the key fact that motivates our use of Haskell as an implementation language — by representing a fuzzy subset by its membership function, a functional language allows us to manipulate such sets/functions with ease. We shall thus use the notion of a fuzzy subset and that of a (fuzzy) characteristic function interchangeably. In particular, if we have a fuzzy subset F of a set X then we shall denote X as the *domain* of F.

To give a concrete example, consider the problem of determining whether a company is profitable based, say, on the profit expressed as a percentage of total costs. Using normal set theory, given a set of percentages, P, we would have to determine an arbitrary cut-off point at and above which we would consider profitable, 15% say (see Figure 1). So we can define *profitable* $\subseteq P$ as:

$$\mathit{profitable} = \{p \mid p \in P \wedge p \geq 15\}$$

This means however that a profit of 14.9% is *not* considered profitable, which is somewhat counter-intuitive considering its proximity to the cut-off point.

Contrast this with a fuzzy definition of *profitable* (see Figure 2). As before, profits above 15% are considered definitely profitable and those below 0% definitely *not* profitable; however between these two figures the degree of profitability increases linearly. For example, a profit of 10% can be regarded as profitable to a degree of 0.67 (i.e., $\mu_{\mathit{profitable}} = 0.67$) and a profit of 14.9% is profitable to a degree of 0.993.

As functions and fuzzy subsets are identical, we represent fuzzy subsets in Haskell as a function from some domain to the fuzzy truth value set. We define the following type synonym:

```
type Fuzzy a = a -> Double
```

A number of functions representing the shapes of common fuzzy subsets are provided (see Figure 3). For instance, `up` has the following definition:

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

The other subsets in Figure 3 can be defined similarly. We can now define the fuzzy subset *profitable* as follows:

```
type Percentage = Double

profitable :: Fuzzy Percentage
profitable = up 0 15
```

Membership testing is then merely function application. For example:

```
Profit> profitable 10
0.666667
```

3.1 The Domain, Support and Fuzziness of a Fuzzy Subset

Knowing the domain of a fuzzy subset is necessary when defuzzifying it (see Section 3.4) and for evaluating its fuzziness (see below). We can also define fuzzy numbers in terms of their fuzziness (see Section 3.3) for which again we need to know the domain over which we are approximating.

Both discrete and continuous domains are represented using ordered lists (in the latter case we only have an approximation). We introduce the type synonym:

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

The 'dot-dot' method of defining lists can be used to define domains in a compact and easily-understandable way. So, for example, we can represent the domain of *profitable*, which is the range $[-10, 30]$ as the list `[-10..30]`.

The *support*, which we shall denote as $\sigma(B)$ (also written as $\text{supp}(B)$) of a fuzzy subset B is the set of those members of its domain, A say, which are in the fuzzy subset with non-zero truth value, i. e.

$$\sigma(B) = \{\mu_B(x) \neq 0 \mid x \in A\}$$

For example, if we take the domain of *profitable* as $[-10, 30]$ then its support is $(0, 30] = \{x \mid 0 < x \leq 30\}$. This has a simple translation into Haskell:

```
supp :: Domain a -> Fuzzy a -> [a]
supp dom f = filter (\x -> f x > 0) dom
```

For example, we can evaluate the support of `profitable` (defined above) *viz*:

```
Profit> supp [-10..30] profitable
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0,
 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0,
 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0]
```

The *fuzziness* ν of a fuzzy subset is the degree to which the values of its membership function cluster around 0.5. The function δ which measures the distance of a truth value to the nearest extreme, 0 or 1:

$$\begin{aligned}\delta(x) &= x, \ \textbf{if}\ x < 0.5 \\ &= 1 - x, \ \textbf{otherwise}\end{aligned}$$

For example $\delta(0.3) = 0.3$, $\delta(0.8) = 0.2$ and $\delta(0) = \delta(1) = 0$. If the domain of our fuzzy subset B is a continuous range, $[a, b]$ say, then we can define ν as:

$$\nu(B) = \frac{2}{b-a}\int_a^b \delta(\mu_B(x))\, dx$$

If the domain is a discrete set of points, $x_1, \ldots, x_n$ say, then the integral becomes a summation:

$$\nu(B) = \frac{2}{n}\sum_{i=1}^{n} \delta(\mu_B(x_i))$$

For example, the fuzziness of *profitable* (again over $[-10, 30]$) is 0.1875. Note that for any crisp set, A, in which the membership function returns only the values 0 or 1, $\nu(A) = 0$ as $\forall x \in A \;.\; \delta(\mu_A(x)) = 0$. Translating the above into Haskell yields the following function:

```
fuzziness :: Domain a -> Fuzzy a -> Double
fuzziness dom f = (2.0 / size_dom) * sum (map (delta.f) dom)
    where
        size_dom = fromInt (length dom)
        delta x
            | x < 0.5   = x
            | otherwise = 1.0 - x
```

For example, we can calculate the fuzziness of `profitable`, *viz*:

```
Profit> fuzziness [-10..30] profitable
0.182114
```

The value that Haskell returns is only an approximation, of course. A better approximation can be obtained by using a domain with more elements, e. g.:

```
Profit> fuzziness [-10,-9.75..30] profitable
0.186335
```

3.2 Fuzzy Subset Operations

Standard set operations — such as union, intersection and complement — can be used with fuzzy subsets. For fuzzy subsets, A, B of a set X, we have:

$$\begin{aligned}A \cup B &= \{\langle x, \mu_A(x) \vee \mu_B(x)\rangle \mid x \in X\} \\ A \cap B &= \{\langle x, \mu_A(x) \wedge \mu_B(x)\rangle \mid x \in X\} \\ A^c &= \{\langle x, \neg\mu_A(x)\rangle \mid x \in X\}\end{aligned}$$

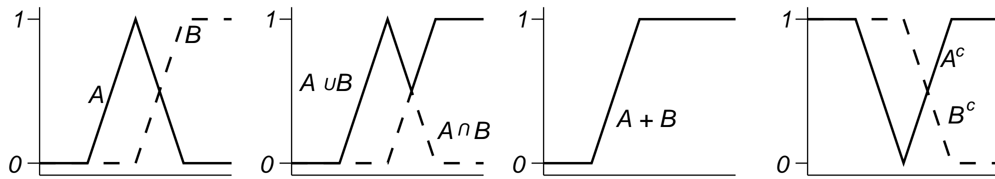

Fig. 4. Operations on fuzzy subsets.

This can be seen graphically in Figure 4, where the logical connectives are defined using Zadeh's method. A slightly unorthodox operation is addition defined as:

$$A + B = \{\langle x, \mu_A(x) + \mu_B(x)\rangle \mid x \in X\}$$

This leads to fuzzy subsets whose membership function returns values outside the range $[0, 1]$. This operation is generally only used in fuzzy systems (see below) where the resultant set is only used as an intermediate value and will be defuzzified (see Section 3.4) to yield a typical value.

If fuzzy subsets are Haskell functions, then the fuzzy subset operators are higher-order functions. If we look at the definition of intersection, for example, we see that we can regard it as a way of defining logical conjunction over sets. This concept holds for both fuzzy *and* crisp sets. Taking this to its logical conclusion we have:

```
instance (Logic b) => Logic (a -> b) where
    true        = \x -> true        -- everything
    false       = \x -> false       -- empty
    f && g      = \x -> f x && g x -- intersection
    f || g      = \x -> f x || g x -- union
    not f       = \x -> not (f x)  -- complement
```

This instance represents a generalized set, where `true` represents the set that everything is a member of and `false` is the empty set. If `true` is an identity for the `&&` over the type `b` then `true` it also an identity for `&&` over the type `a -> b`, and similarly for `false` and `||`.

In the context of fuzzy subsets, that is the type `Fuzzy a` (which in turn is the type `a -> Double`), true is the fuzzy subset, T say, with membership function $\mu_T(x) = 1$ and false is the fuzzy subset, F say, with membership function $\mu_F(x) = 0$. The function `true` remains the identity of `&&` and `false` the identity of `||`. We also need to be able to perform addition on fuzzy subsets. This is done by making the type `a -> b`, which remember is a generalisation of the type `Fuzzy a` a member of the `Num` class (which is used to overload the numeric operators `+`, `-`, etc.):

```
instance (Num a, Num b) => Num (a, b) where
instance (Num b) => Num (a -> b) where
    f + g           = \x -> f x + g x
    f * g           = \x -> f x * g x
    abs f           = \x -> abs (f x)
    signum f        = \x -> signum (f x)
```

```
    negate f        = \x -> negate (f x)
    fromInteger i   = \x -> fromInteger i
```

We will also find it useful to use the operators of the `Logic` class over tuples, for instance in the shower controller described in Section 4.1.1 which groups its output variables in tuples. This is done pointwise, e. g., for pairs we have:

```
instance (Logic a, Logic b) => Logic (a, b) where
    true                 = (true, true)
    false                = (false, false)
    (a, b) && (a', b')   = (a && a', b && b')
    (a, b) || (a', b')   = (a || a', b || b')
    not (a, b)           = (not a, not b)
```

We also declare tuples to be instances of the `Num` class in a similar manner.

3.3 Hedges and Fuzzy Numbers

Just as adjectives such as profitable can be qualified by terms such as *very* and *somewhat*, so can fuzzy subsets. Terms such as these, known as *hedges* alter the membership function by intensifying it (normally by raising it to a power greater than 1) in the case of *very* and similar terms such as extremely, or diluting it (normally by raising it to a power between 0 and 1) in the case of *somewhat*. Usually we have:

$$\mu_{very\ F}(x) = \mu_F(x)^2$$
$$\mu_{somewhat\ F}(x) = \mu_F(x)^{1/2}$$

The effect of *very* and *somewhat* on *profitable* can be seen in Figure 5. We see that a profit of 10% is *profitable* with truth value 0.67, *very profitable* by truth value 0.44, and *somewhat profitable* by degree 0.82.

In Haskell, we represent hedges as higher-order functions. We first define a generic hedge which will raise the value of a function to a specified power:

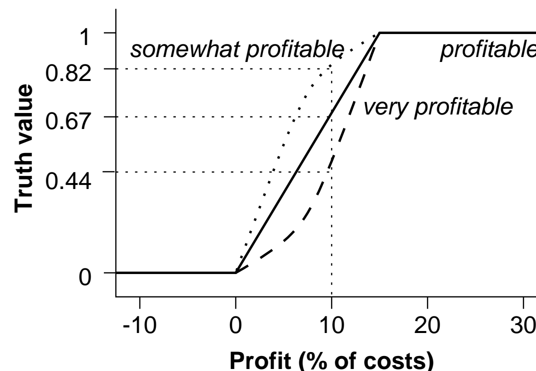

Fig. 5. *Very profitable* and *Somewhat profitable.*

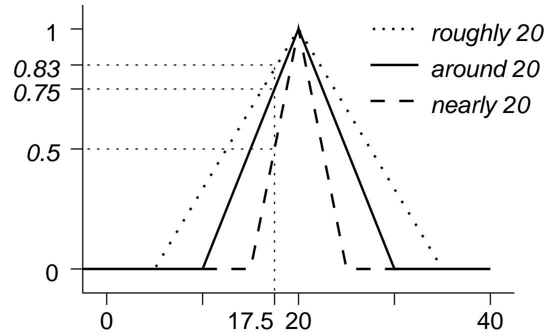

Fig. 6. Fuzzy approximations to 20.

```
hedge :: Double -> Fuzzy a -> Fuzzy a
hedge p f x = if fx == 0 then 0 else fx ** p
    where fx = f x
```

We can then define more specific hedges as follows:

```
very, extremely, somewhat, slightly :: Fuzzy a -> Fuzzy a
very        = hedge 2
extremely   = hedge 3
somewhat    = hedge 0.5
slightly    = hedge (1 / 3)
```

The user is free to redefine these functions with different numbers if they want, of course. An example of these in use, using the same sets and definitions in Figure 5:

```
Profit> very profitable 10
0.444444
Profit> somewhat profitable 10
0.816497
```

Hedges can also be used to approximate numbers by converting them into fuzzy subsets (also known as *fuzzy numbers* in this context) using such terms as *around* 20, *roughly* 20 and *nearly* 20. One typical way of defining these subsets is by symmetrical triangular fuzzy subsets, centred on the number, c say, that we are approximating and with base of width $2w$. The membership function of this set is thus:

$$\begin{array}{rcll} \mu(x) & = & 1 - \frac{|x-c|}{w} & \textbf{if } c - w \le x \le c + w \\ & = & 0, & \textbf{otherwise} \end{array}$$

The tighter the approximation we want, the less fuzzy the fuzzy subset is, and hence the smaller the base of the triangular fuzzy subset is. In general, *roughly* is a looser approximation than *around* which in turn is looser than *nearly*.

For example, consider the fuzzy numbers in Figure 6, which approximate 20 over the domain $[0, 40]$ using triangular fuzzy subsets centred on 20. Here we see that *nearly 20* has a base of length 5 and a fuzziness of 0.125; *around 20* has a base of length 10 and a fuzziness of 0.25; and *roughly 20* has a base of length 15 and a

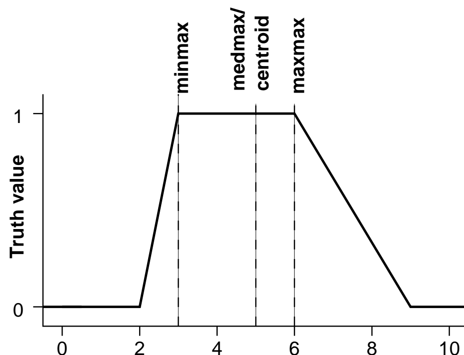

Fig. 7. Defuzzifying a fuzzy subset

fuzziness of 0.375. So, for example, 17.5 is *nearly* 20 with truth vale 0.5, *around* 20 with truth value 0.75 and *roughly* 20 with truth value 0.83.

As with hedges, to implement fuzzy numbers in Haskell we define a generic fuzzy number function, which approximates a number on a specific domain by a triangular fuzzy subset (see Figure 3) of specified fuzziness:

```
approximate :: Double -> Double -> Domain Double -> Fuzzy Double
approximate fuzziness n dom = tri (n - hw) (n + hw)
    where hw = fuzziness * (ub dom - lb dom)
```

We now define the fuzzy number generators `near`, `around` and `roughly` as:

```
near, around, roughly :: Double -> Domain Double -> Fuzzy Double
near    = approximate 0.125
around  = approximate 0.25
roughly = approximate 0.375
```

This leads to the same sets as in Figure 6 if we approximate 20 over the domain $[0, 40]$. For example:

```
Profit> near 20 [0..40] 17.5
0.5
Profit> roughly 20 [0..40] 17.5
0.833333
Profit> around 20 [0..40] 17.5
0.75
```

3.4 Defuzzification

In a real-world situation, we often need a concrete value rather than a fuzzy subset. The process of extracting a typical value from a fuzzy subset is known as *defuzzification* and there are many methods for doing this. Two such methods are finding the *centroid* (or centre of gravity) of a fuzzy subset, or finding the *maxima* of a fuzzy subset and returning a member of this set.

If we have a fuzzy subset A with membership function μ_A over a domain X then the centroid of A is defined as:

$$\frac{\int_X x\mu_A(x)\ dx}{\int_X \mu_A(x)\ dx}$$

if X is a continuous domain. If X is discrete then the centroid is defined as:

$$\frac{\sum_X x\mu_A(x)}{\sum_X \mu_A(x)}$$

The latter is the definition we use in our implementation. We define the `centroid` function as::

```
centroid :: Domain Double -> Fuzzy Double -> Double
centroid dom f = (sum (zipWith (*) dom fdom)) / (sum fdom)
    where fdom = map f dom
```

For example, the centroid of the trapezoid fuzzy subset in Figure 7 can be evaluated *viz*

```
Profit> centroid [0..10] (trap 2 3 6 9)
5.06667
```

The maxima of a fuzzy subset A over a domain X is defined as the set $maxima(A)$ such that:

$$\forall m \in maxima(A)\ .\ \forall x \in X\ .\ \mu_A(m) \geq \mu_A(x)$$

This can be implemented using the following function:

```
maxima :: Ord a => Domain a -> Fuzzy a -> [a]
maxima dom f = maxima' dom []
    where
        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)
```

We then typically defuzzify A by returning the minimum, the median or the maximum of $maxima(A)$:

```
minmax, medmax, maxmax :: Ord a => Domain a -> Fuzzy a -> a
minmax dom f = minimum (maxima dom f)
maxmax dom f = maximum (maxima dom f)
medmax dom f = median (maxima dom f)
    where
    median ms = head (drop (length ms `div` 2) (qsort ms))
    qsort []     = []
    qsort (x:xs) = qsort [y | y <- xs, y <= x] ++ [x] ++
                   qsort [y | y <- xs, y > x]
```

Defuzzifying the fuzzy subset in Figure 7 using these three methods we get:

```
Profit> minmax [0..10] (trap 2 3 6 9)
3.0
Profit> medmax [0..10] (trap 2 3 6 9)
5.0
Profit> maxmax [0..10] (trap 2 3 6 9)
6.0
```

3.5 An Example — Fuzzy Database Queries

The linguistic nature of fuzzy subsets make them ideal in database enquiries. In a functional language this is akin to applying a filter to a list of information. We define a variant of the standard filter function, which takes a *fuzzy* predicate (i. e. a function which returns a fuzzy truth value) and returns those members of the list that satisfy the predicate to a non-zero degree, along with the degree to which they satisfy the predicate:

```
ffilter :: Fuzzy a -> [a] -> [(a, Double)]
ffilter p xs = filter ((/=) 0 . snd) (map (\x -> (x, p x)) xs)
```

Referring back to our profit example, based originally on an example in (Negoita, 1985), suppose we have the following module:

```
module Profit where

import Prelude hiding ((&&), (||), not, and, or, any, all)
import Fuzzy

type Percentage = Double
type Sales      = Double -- thousands of pounds
type Company    = (String, Sales, Percentage)

sales :: Company -> Sales
sales (_, s, _)  = s

profit :: Company -> Percentage
profit (_, _, p) = p

percentages :: [Percentage]
percentages = [-10..30]

profitable :: Fuzzy Percentage
profitable = up 0 15

high :: Fuzzy Sales
high = up 600 1150
```

```
companies :: [Company]
companies = [("A", 500, 7),   ("B", 600, -9),  ("C", 800, 17),
             ("D", 850, 12),  ("E", 900, -11), ("F", 1000, 15),
             ("G", 1100, 14), ("H", 1200, 1),  ("I", 1300, -2),
             ("J", 1400, -6), ("K", 1500, 12)]
```

So, we have a list of companies, functions to extract their profit and sales, and fuzzy subsets `profitable` of `Percentage` (using the same definition as before) and `high` of `Sales`. To extract all the profitable companies from `companies`, we first define the fuzzy predicate `p1`:

```
p1 co = profitable (profit co)
```

and `ffilter` it over `companies`, *viz*:

```
Profit> ffilter p1 companies
[(("A",500.0,7.0),  0.466667), (("C",800.0,17.0),1.0),
 (("D",850.0,12.0), 0.8),      (("F",1000.0,15.0),1.0),
 (("G",1100.0,14.0),0.933333), (("H",1200.0,1.0),0.0666667),
 (("K",1500.0,12.0),0.8)]
```

So, of the original 11 companies, 7 are considered profitable with `C` and `F` being the most profitable. Profitability by itself might not be enough — we may also want high sales. Defining:

```
p2 co = profitable (profit co) && high (sales co)
```

we can then find all profitable companies with high sales:

```
Profit> ffilter p2 companies
[(("C",800.0,17.0),0.363636),  (("D",850.0,12.0),0.454545),
 (("F",1000.0,15.0),0.727273), (("G",1100.0,14.0),0.909091),
 (("H",1200.0,1.0),0.0666667), (("K",1500.0,12.0),0.8)]
```

Six companies satisfy the predicate, with `G` satisfying it the most. We can use hedges to tighten or loosen the conditions, for example, defining

```
p3 co = somewhat profitable (profit co) && very high (sales co)
```

we can find those companies which have very high sales and somewhat profitable:

```
Profit> ffilter p3 companies
[(("C",800.0,17.0),0.132231),  (("D",850.0,12.0),0.206612),
 (("F",1000.0,15.0),0.528926), (("G",1100.0,14.0),0.826446),
 (("H",1200.0,1.0),0.258199),  (("K",1500.0,12.0),0.894427)]
```

Here the increased emphasis on sales, and decreased emphasis on profitability means that company `K` now satisfies the predicate we pass to `ffilter` to the highest degree.

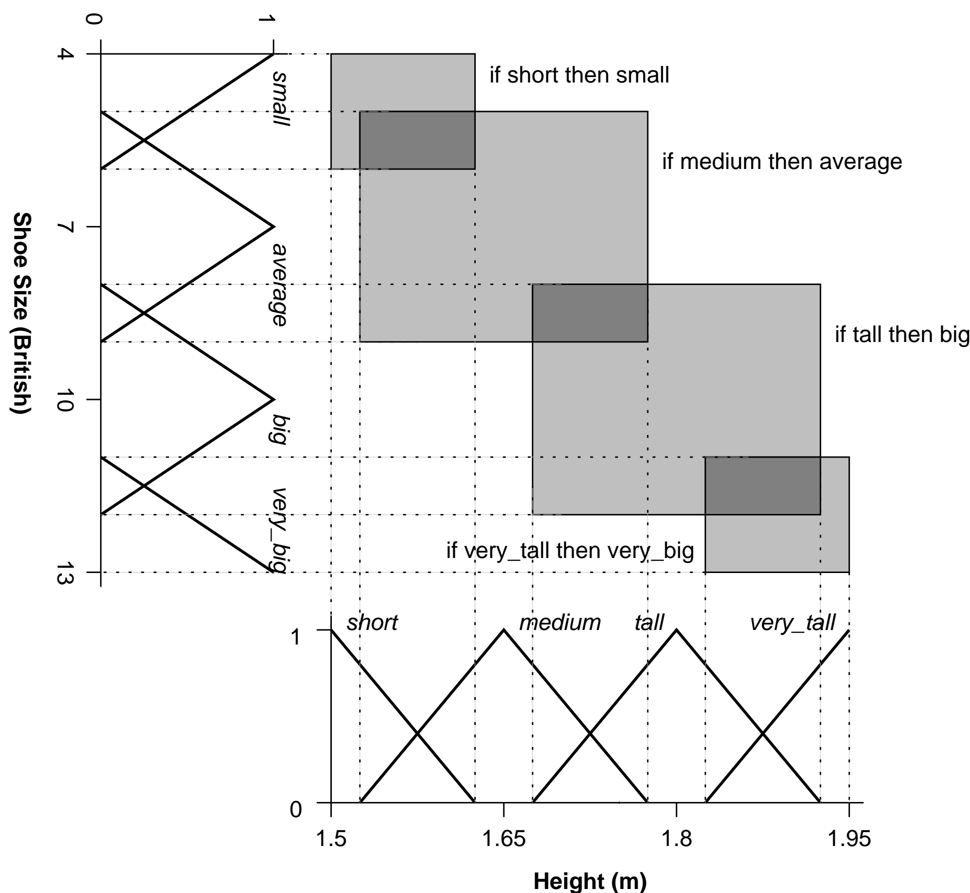

Fig. 8. The fuzzy rule base for the height → shoe size expert system

4 Fuzzy Systems

Expert Systems (Russel & Norvig, 1995) are used to model real-world situations in many areas of expertise. One common way of implementing these systems is as a set of *rules* and an *inference engine* which manages these rules. Rules are composed of two parts: an *antecedent*, which is a logical expression; and a *consequent* which is an action which is performed when the antecedent is true. When this happens we say that the rule *fires*.

As a simple example, consider predicting the shoe size, using British shoe sizes, of a man given his height in metres. In a standard expert system we might have rules like:

```
if 1.65 <= height & height <= 1.72 then shoe_size := 9
```

These rules are absolutes — if and only if the antecedent holds will the action be fired and fired completely.

In a rule-based fuzzy system, the antecedent is a fuzzy logic expression the value of which dictates the degree to which the action fires, the action being the assignment

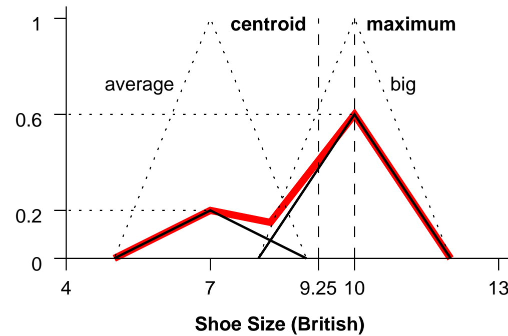

Fig. 9. Weighting, adding and defuzzifying the rules for a height of 1.75m

of a variable to a fuzzy subset. If we have a rule such as `if` p `then` a `:=` F then a is assigned to the fuzzy subset F' where F' is linearly weighted by the value of p and has membership function $\mu_{F'}(x) = p\mu_F(x)$. This can be extended to multiple variable assignments. Note that if the value of the antecedent is 0, then the membership function of the consequent fuzzy subset will be constantly 0 (the empty set) and we regard the rule as not having been fired. In our shoe size example, our rules are:

```
if height is short then shoe_size := small
if height is medium then shoe_size := average
if height is tall then shoe_size := tall
if height is very_tall then shoe_size := very_big
```

Here `is` serves as a membership test for `height`. These rules can be thought of as forming *patches* (see Figure 8) with the larger the patch the fuzzier the rule (Kosko, 1994). More input variables require more dimensions to the patches.

As can be seen, these patches overlap, which in practical terms means that more than one rule can fire, i. e., we have more than one possible assignment to `shoe_size`. Rather than selecting one of the possible assignments to a we select them *all*, combining the subsets into one set using an operation such as union or addition. Addition has the property that, unlike union, when combining many sets the membership function of the result doesn't approach the constant function 1. Also all the sets that are part of the addition contribute to the final result, whereas in the case of union large sets (measured by both their support and their height (truth values)) subsume smaller ones.

Once we have combined all the resultant sets, we then defuzzify them (see Section 3.4) to obtain a final result. For instance, if we have a height of 1.75m then this is tall to degree 0.6 and medium to degree 0.2. If we weight the relevant consequents, sum the sets and defuzzify using the centroid method we obtain an estimated shoe size of $9\frac{1}{4}$, while defuzzifying with any of the maxima methods yields a shoe size of 10, since 10 is the only element of the resultant fuzzy subset which yields the largest truth value, in this case 0.6 (see Figure 9). Of course, this is a very simple example. More complex ones can be found in Section 4.1.

We introduce a new operator `==>`, which has the leastmost binding, to the `Logic` class:

```
infix 0 ==>

class Logic a where
    (==>) :: Double -> a -> a          -- other defs as before
```

This operator linearly weights its right-hand side by the value on its left-hand side. On fuzzy values, it is simply multiplication:

```
instance Logic Double where
    (==>) = (*)                        -- other defs as before
```

There are a number of definitions over `Bool`. One such definition is:

```
instance Logic Bool where
    w ==> False = False
    w ==> True  = w > 0.5              -- other defs as before
```

The `==>` function used over fuzzy truth values is useful in its own right as a fuzzy if-then function; an example of its use can be seen in Section 4.1.2. However its major use is to represent a rule in a fuzzy rulebase, where we normally expect the value on the RHS of the operator to be a fuzzy subset or a tuple of such sets. On fuzzy subsets, this operator has the definition:

```
instance (Logic b) => Logic (a -> b) where
    w ==> f = \x -> w ==> f x          -- other defs as before
```

and on tuples we weight each element of the tuple individually, e.g., for pairs we have:

```
instance (Logic b) => Logic (a -> b) where
    w ==> (a, b) = (w ==> a, w ==> b) -- other defs as before
```

The LHS of the `==>` is thus the antecedent of the rule and the RHS of the rule is the consequent. The result of the function is the consequent linearly weighted by the antecedent, which will usually be the result of evaluating fuzzy logic expression.

To combine the weighted subsets we define a function which takes a list of subsets and a function to combine (two of) them with, and returns the result of combining all the weighted subsets. We thus just have:

```
rulebase :: Logic a => (a -> a -> a) -> [a] -> a
rulebase = foldr1
```

Note that we can't apply `rulebase` to the empty list, but this would imply we had an empty set of rules. The resultant set can then be defuzzified using one the defuzzifying functions from Section 3.4.

Putting this all together, we have the following Haskell module which implements our shoe-size expert system from above:

```
module Shoe where

import Prelude hiding ((&&), (||), not, and, or, any, all)
import Fuzzy

type Height     = Double -- Metres
type ShoeSize   = Double -- British size

sizes :: Domain ShoeSize
sizes = [4, 4.5..13]

short, medium, tall, very_tall :: Fuzzy Height
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 :: Fuzzy ShoeSize
small       = down 4  6
average     = tri  5  9
big         = tri  8 12
very_big    = up  11 13

-- calculate the shoe size from a given height
shoe_size :: Height -> ShoeSize
shoe_size h = centroid sizes (
    rulebase (+) [
        short h     ==> small,
        medium h    ==> average,
        tall h      ==> big,
        very_tall h ==> very_big])
```

Consider the use of the `rulebase` function inside the `shoe_size` function. Its first argument is `+`, i. e., we are using fuzzy subset addition to combine the weighted subsets. Its second argument is the set of rules, written using the `==>` operator. During evaluation of the `rulebase` function, each of these rules will be evaluated, giving the required weighted set, which will all then be combined, in this case using `+`. This set is then defuzzified using the `centroid` function over the domain `sizes`.

4.1 Further Examples

4.1.1 Controlling a Shower

Consider the problem of controlling a shower (for Information Technology, 1996). We wish to get the temperature to between 34°C and 38°C and the flow of the water between 11 l/min and 13 l/min. To do this we have two taps, one hot and

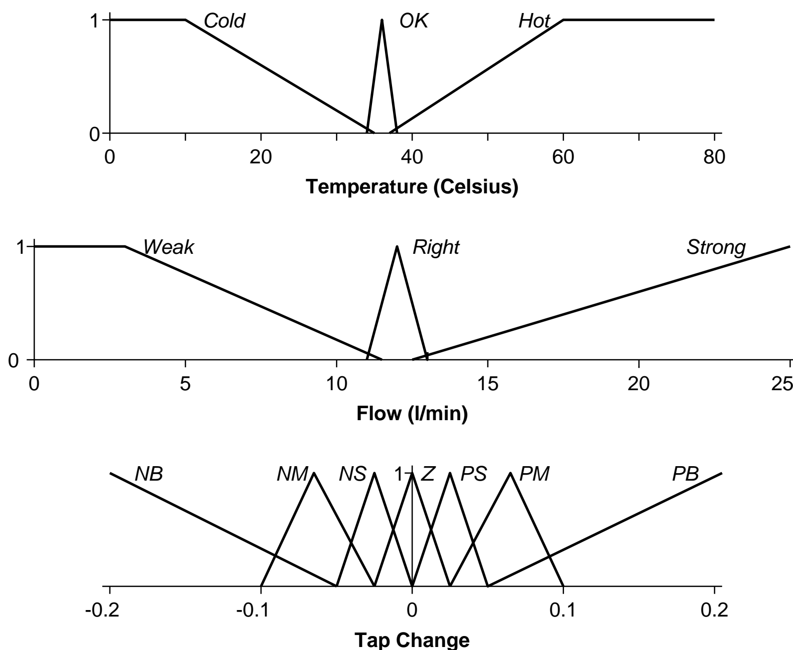

Fig. 10. Fuzzy subsets of temperature, flow and tap change

one cold, which take values between 0 (fully off) and 1 (fully on). We divide the temperature into the fuzzy subsets `hot`, `ok` and `cold`; the flow into the fuzzy subsets `weak`, `right` and `strong`; and the possible tap changes (ranging from -0.2 to 0.2) into seven fuzzy subsets: `pb` (big positive change), `pm` (medium positive change), `ps` (small positive change), `z` (zero change), `ns` (small negative change), `nm` (medium negative change) and `nb` (big negative change). These fuzzy subsets can be seen in Figure 10.

Unlike our shoe size example, the shower is not meant to be a one-use function but rather to be continually iterated until the temperature and the flow are in the correct range. So we are continually making changes (with suitable gaps in between these changes to let the shower settle into its new settings) until the water becomes acceptable. We have the following system (note that these are not the original sets used in the Fuzzy CLIPS example, which used curved rather than polygonal fuzzy sets, and hence we have tweaked the numbers to get a better performance):

```
module Shower where

import Prelude hiding ((&&), (||), not, and, or, any, all)
import Fuzzy

type Temp = Double
type Flow = Double
type Change = Double
```

```
cold, ok, hot :: Fuzzy Temp
cold    = down 15 36
ok      = tri  32 40
hot     = up   36 75

weak, right, strong :: Fuzzy Flow
weak    = down  0 12
right   = tri   9 15
strong  = up   12 25

nb, nm, ns, z, ps, pm, pb :: Fuzzy Change
nb  = down (-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  = up     0.05    0.2

change_valves :: (Temp, Flow) -> (Change, Change)
change_valves (temp, flow) = (defuz hv, defuz cv)
    where
    defuz = centroid [-0.2, -0.195..0.2]
    (hv, cv) = rulebase (+) [
                    cold temp && weak flow      ==> (pm, z),
                    cold temp && right flow     ==> (pm, z),
                    cold temp && strong flow    ==> (z, nb),
                    ok temp && weak flow        ==> (ps, ps),
                    ok temp && strong flow      ==> (ns, ns),
                    hot temp && weak flow       ==> (z, pb),
                    hot temp && right flow      ==> (nm, z),
                    hot temp && strong flow     ==> (nb, z)]
```

4.1.2 Pricing Goods

The fact that fuzzy logic is inherently contradictory, that is we have truth values which are non-zero and whose negation is also non-zero, is useful in decision making processes where the decisions we have to make are based on conflicting demands or requirements. Fuzzy logic can be used to resolve these contradictions in a natural, simple and efficient way.

Consider the problem of pricing goods (Cox, 1994). The price should be as high as possible to maximise takings but as low as possible to maximise sales. We also want to make a healthy profit, say a 100% mark-up on the cost price. Then we have to consider what the competition is charging. We can formalise these requirements as rules:

1. Our price must be high.
2. Our price must be low.
3. Our price must be around 2 × manufacturing costs (i. e., a 100% mark-up).
4. If the competition price is not very high then our price must be around the competition price (we don't want to indulge in a price war).

A boolean system may have difficulties trying to resolve the requirements that the price must be high *and* low, not to mention the other two requirements, but a fuzzy system has no such difficulties.

Suppose possible prices are in the range £15 to £35. We define fuzzy subsets `high` and `low` on this range, *viz*:

```
type Price = Double -- Pounds Sterling

prices :: Domain Price
prices = [15.00, 15.50 .. 35.00]

high, low :: Fuzzy Price
high = up 15.00 35.00
low  = not high
```

So if we want a price that is high and low (Rules 1 and 2) then we can calculate this by taking the intersection of `high` and `low` and defuzzifying the resultant set to get a typical value, *viz*:

```
our_price = centroid prices (high && low)
```

Evaluating `our_price` we get:

```
Prices> our_price
25.0
```

Rule 3 suggests that we can approximate the price by a fuzzy number centred on 2 × manufacturing costs. Taking the manufacturing costs as a parameter to `our_price` and combining this with what we have so far, we define

```
our_price' man_costs =
    centroid prices (high && low &&
                     around (2.0 * man_costs) prices)
```

Assuming manufacturing costs of £13.25, say, we have:

```
Prices> our_price' 13.25
26.252
```

Rule 4 is a conditional rule. The more that the competition price is not very high, the more it affects the calculation of our price. Using the `==>` operator and taking the competition price as another parameter, we get:

```
our_price'' man_costs comp_price =
    centroid prices (high && low &&
                     around (2.0 * man_costs) prices &&
                     ((not.very high) comp_price ==>
                            around comp_price prices))
```

Assuming the same manufacturing costs as before and a competition price of £29.99 we have:

```
Prices> our_price'' 13.25 29.99
28.5893
```

So our final retail price is £28.59.

5 Conclusion

We have introduced and 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 fuzzy systems, used in a variety of control and decision making problems, can be implemented in a functional language in a natural and efficient way.

References

Aptronix Ltd. (1992a). *Focusing system*. http://www.aptronix.com/fuzzynet/applnote/focusing.htm.

Aptronix Ltd. (1992b). *Washing machine*. http://www.aptronix.com/fuzzynet/applnote/wash.htm.

Aptronix Ltd. (1996). *Fuzzy java*. http://www.aptronix.com/fuzzynet/applnote/java.htm.

Cox, Earl. (1994). *The fuzzy systems handbook*. AP Professional.

Eklund, Patrik, & Kwalonn, Frank. (1992). Neural fuzzy logic programming. *Ieee Transactions on Neural Networks*, **3**(5), 815–818.

Fodor, János, & Roubens, Marc. (1994). *Fuzzy preference modelling and multicriteria decision support*. Kluwer Academic Press.

for Information Technology, NRC-CNC Institute. (1996). *Fuzzy CLIPS*. WWW: http://ai.iit.nrc.ca/ fuzzy/ fuzzy.html.

Hall, Cordelia, Hammond, Kevin, Peyton Jones, Simon, & Wadler, Philip. (1996). Type classes in Haskell. *Acm Transactions on Programming Languages and Systems*, **18**(2), 109–138.

Horvath, J.M. (1988). A fuzzy set model of learning disability. *Pages 345–382 of:* Zétényi, Tamás (ed), *Fuzzy sets in pyschology*. Advances in Pyschology, no. 56. North-Holland.

Jones, Mark. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, **5**(1).

Kaufmann, Arnold. (1975). *Introduction to the theory of fuzzy subsets*. Vol. 1. Academic Press.

Kosko, Bart. (1994). *Fuzzy thinking*. Flamingo.

Ltd., Logic Programming Associates. (1997). *FLINT toolkit*. WWW: http:// www.lpa.co.uk/ fln.html.

Łukasiewicz, Jan. (1967a). On the notion of possibility/On three-valued logic. *Pages 15–18 of:* McCall, Storrs (ed), *Polish logic 1920–1939*. Oxford University Press. Appeared originally under the titles 'O pojęciu możliwości' and 'O logice trojwartosciowej' in *Ruch Filozoficzny* 5 (1920), pp 169–171.

Łukasiewicz, Jan. (1967b). Philosophical remarks on many-valued systems of propositional logic. *Pages 40–65 of:* McCall, Storrs (ed), *Polish logic 1920–1939*. Oxford University Press. Appeared originally under the title 'Philosophische Bemerkungen zu mehrwertigen Systemen des Aussagenkalküls' in *Comptes rendus des séances de la Societé des Sciences et des Lettres de Varsovie*, Cl. iii, 23 (1930), pp 51–77.

Mamdani, E.H., & Assilian, S. (1975). An experiment in linguistic synthesis with a fuzzy logic controller. *International Journal of Man-Machine Studies*, 1–13.

Martin, T.P., Balwdin, J.F., & Pilsworth, B.W. (1987). The implementation of FProlog — a fuzzy Prolog interpreter. *Fuzzy Sets and Systems*, **23**, 119–129.

Negoita, C.V. (1985). *Expert systems and fuzzy systems*. The Benjamin/Cummings Publishing Company.

Peterson, John, & Hammond, Kevin (editors). 1997 (April). *The Haskell 1.4 report*. http://haskell.org/report/.

Peyton Jones, Simon, Jones, Mark P., & Meijer, Erik. (1997). Type classes: exploring the design space. *Proceedings of the Haskell Workshop, Amsterdam, June 6*.

Ross, Timothy J. (1995). *Fuzzy logic with engineering applications*. McGraw-Hill.

Russel, Stuart, & Norvig, Peter. (1995). *Artificial Intelligence — a modern approach*. Prentice Hall.

Tanaka, Kazuo. (1997). *An introduction to fuzzy logic for practical applications*. Springer-Verlag. First published in Japanese, 1991.

Thompson, Simon. (1996). *Haskell: The craft of functional programming*. Addison-Wesley.

Wang, Li-Win. (1994). *Adaptive fuzzy systems and control — design and stability analysis*. Prentice Hall.

Yan, Jun, Ryan, Michael, & Power, James. (1994). *Using fuzzy logic*. Prentice Hall.

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, **8**, 338–353.

Zadeh, L. A. (1973). Outline of a new approach to the analysis of complex systems and decision processes. *Ieee Transactions on Systems, Men and Cybernetics*, **3**, 28–44.

Zimmermann, H.-J. (1991). *Fuzzy set theory — and its applications*. Kluwer Academic Publishers.