Appendix G

Other Fuzzy Unification Work

In the course of this work, we searched the literature for "*fuzzy unification*" and "*fuzzy logic programming*," and variations thereof. Our first observation in pursuing this interest has been that, unlike existing Prolog languages (and other technology that relies on standard $\mathcal{FOT}$ unification), not all the fuzzy $\mathcal{LP}$ languages that have been proposed and/or implemented (even if only as prototypes) agree on the same fuzzy $\mathcal{FOT}$ unification operation. We have looked at some of the most prominent among those existing in an attempt to characterize their fuzzy unification operations. We next summarize some essential points that we understood of these variations on fuzzy unification from our perspective. We will proceed succintly, as it would be presumptuous of us to give an exhaustive recap of research in Fuzzy $\mathcal{LP}$. Again, our perspective is not so much Fuzzy $\mathcal{LP}$ as it is that of understanding its fuzzification of the lattice-theoretic operations on $\mathcal{FOT}$s such as $\mathcal{FOT}$ unification, which an essential part on any $\mathcal{LP}$ system.

Before we focused on M. Sessa's fuzzy $\mathcal{FOT}$ unification algorithm, which we present in Chapter 2,[1] we looked at other work which we recall here. We also discuss our reasons for our choice to follow Sessa's approach as opposed to fuzzy Datalog or edit-distance $\mathcal{FOT}$ matching we describe next.

Fuzzy Datalog unification

Among earlier frequently cited works related to fuzzy $\mathcal{FOT}$ unification is [24], where the title leads one to expect a fuzzy term unification in a fuzzy $\mathcal{LP}$ language. It is not quite that, however, as it restricts solving similarity equations over word symbols tolerating some imprecise matches (*i.e.*, a kind of fuzzy Datalog).[2] Such mismatches could come from (possibly accidental) syntactic proximity (*e.g.*, typos, misspellings).[3] The authors propose to accumulate mismatched symbols into what they dub "*clouds*," which represent in effect similarity classes of constant symbols (nullary functors). These clouds are given a measure of "similarity" computed as the meet of those of the components—which they propose to conceive as a "cost" of how much it deviates from a perfect match (which itself has zero cost, since perfect).

[1] See Section 2.2.1.

[2] It is only acknowledged in the very last sentence of the paper, that the authors were yet "*aiming at extending this algorithm to the full-fledged algebra of first-order terms*" as future work.

[3] Or presumably, although they only mention it as a potential further work in their conclusion, from semantic proximity (such as could be specified as fuzzy knowledge in the form of fuzzy similarity matrices).

Note that when the only non-variable terms defined are constants, the rules of Figure 2.3 accumulate all constant mismatches as unresolved equations. Thus, what constitutes Arcelli *et al.*'s "clouds" are the sets of constants making up the equivalence classes of the reflexive-transitive closure of the relation containing these equations.

While this could be useful as a particular fuzzy extension of Datalog, it does not address issues concerning fuzzy database representation and evaluation issues such as expounded in [41] for (crisp) Datalog. In fact, such fuzzy extensions of Datalog had already been proposed, with a straightforward fix-point semantics extending that of classical Datalog (*e.g.*, [1]). Since it limits itself to fuzzification a Datalog-like $\mathcal{LP}$, the semantics of Arcelli *et al.*'s fuzzy $\mathcal{LP}$ language only considers approximate equations between constant symbols, whose mutual fuzzy proximity is specified as fuzzy "proximity matrices". This early form of fuzzy unification was put to use in the fuzzy $\mathcal{LP}$ language Likelog [22], [23], [25], [26].

Edit-distance fuzzy unification

Arcelli *et al.*'s fuzzy constant unification was later elaborated to work on full $\mathcal{FOT}$s as first exposed in [24] and used in [48], and then again in [94] and [95]. The authors use the same kind of fuzzy unification on constants (*i.e.*, names formalized as symbol strings), but instead of names deemed similar (*i.e.*, in a same "cloud" or similarity class), the more classical notion of *edit distance* is used to evaluate the truth value of a fuzzy name match (normalized over the symbol lengths). Edit distance between two strings is the minimal number of elementary edit actions (deleting a character, inserting a character, or replacing a character for another), in either or both strings necessary to obtain a perfect match [69].[4] This fuzzy unification was put to use in biological genetics analysis with the fuzzy $\mathcal{LP}$ language FURY [48], [95].

In what follows, we use our own formal notation to summarize the essence of FURY-style fuzzy $\mathcal{FOT}$ unification [48], [94], [95]. We represent the empty string as ε, and a non-empty string as a dot-separated sequence of characters ending with the empty string (*e.g.*, `"this"` is represented as $\texttt{t.h.i.s.}\varepsilon$). Given a non-empty string $h.t$, we shall call its first character h its "*head*" and the substring t following it, its "*tail*." The length of a string s is its number of characters denoted $|s|$. That is, the monoid homomorphism:

$$\begin{aligned} |\varepsilon| &\stackrel{\text{def}}{=} 0 \\ |h.t| &\stackrel{\text{def}}{=} 1 + |t|. \end{aligned}$$

Thus, the edit distance $\delta(s_1, s_2)$ between two strings s_1 and s_2 is derived as:[5]

$$\left.\begin{aligned} \delta(\varepsilon, s) &\stackrel{\text{def}}{=} |s| \\ \delta(s, \varepsilon) &\stackrel{\text{def}}{=} |s| \\ \delta(h.t_1, h.t_2) &\stackrel{\text{def}}{=} \delta(t_1, t_2) \\ \delta(h_1.t_1, h_2.t_2) &\stackrel{\text{def}}{=} 1 + \mathbf{min}\{\, \delta(t_1, h_2.t_2), \delta(h_1.t_1, t_2), \delta(t_1, t_2) \,\}, \quad \text{if } h_1 \neq h_2. \end{aligned}\right\} \tag{G.1}$$

[4] It is used most crucially in Internet search keyword matches and DNA sequence alignment and matching.

[5] From which one can easily check that expected properties of a distance are verified by δ such as $\delta(s, s) = 0$, $\delta(s_1, s_2) = \delta(s_2, s_1)$, and $\delta(s_1, s_2) \leq \delta(s_1, s) + \delta(s, s_2)$, for any strings s, s_1, and s_2.

These four defining equations express that the edit distance: (1) from any string to the empty string is the length of this string; (2) between two strings with equal first character, it is the distance between the remaining substrings; (3) otherwise, it is one plus the minimum of the three edit distances between one of the strings and the other string's rest, and between the two strings' rests. The latter is known as the "*Levenshtein distance*" between two strings.[6]

Because edit distance will increase with the lengths of strings, it is convenient to calibrate it over the size of the strings involved; hence the notion of "*normalized edit distance*" δ_N as in:

$$\delta_N(s_1, s_2) \stackrel{\text{def}}{=} \frac{\delta(s_1, s_2)}{\mathbf{max}(|s_1|, |s_2|)}. \tag{G.2}$$

In [48], this notion of (normalized) edit distance between constant symbol strings (including the empty string ε) is extended to an edit distance between $\mathcal{FOT}$ trees. It maps two terms t_1 and t_2 to a non-negative number $\delta(t_1, t_2) \stackrel{\text{def}}{=} m^\sigma_n \in \mathbb{N}$, which denotes the minimal total number m of mismatches (edit actions necessary to go from one to the other), along with two collateral pieces of information:

- σ—a most general variable substitution that may be necessary to resolve matches upon encountering variables in the process of computing this miminal edit distance between the two terms when assimilated to strings which are sequences of the non-punctuation characters that compose their their syntax; and,
- n—a normalization factor computed as the sum of the lengths of the longest of each of pair of symbols from each term as they are matched.

The normalization factor n is a function of the lengths of the terms when a term is seen as the string concatenation in the order they appear of the non-variable and non-punctuation symbols in it. In other words, parentheses, commas, variables, and ε are considered of length 0. Namely,

$$|t| \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } t \text{ is a variable} \\ |f| + \Sigma_{i=1}^n |t_i| & \text{if } t = f(t_1, \ldots, t_n),\ n \geq 0. \end{cases}$$

Compounding two $\mathcal{FOT}$ edit distances m^σ_n and p^θ_q consists in adding them up while composing their substitutions and adding their normalization factors:

$$m^\sigma_n + p^\theta_q \stackrel{\text{def}}{=} (m + p)^{\theta\sigma}_{n+q}. \tag{G.3}$$

In other words, as it sums the numbers of symbol mismatches, it also composes their associated variable substitutions and sums their normalization factors (which depend on the sizes of all the involved symbols). Note that this operation, while commutative in its numerical arguments (which are added), is not commutative in its substitution arguments (which are composed). It could also be defined by composing the substitutions in the other order if wished; but this is simpler.[7]

[6] The Levenshtein distance between two strings has the advantage to apply to strings of differring as well as of equal lengths. This is unlike the *Hamming distance* which is restricted to strings of equal lengths, and defined as the number of disagreeing character positions. This entails, in particular, that the Levenshtein distance between two equal-length strings is always less than or equal to their Hamming distance.

[7] The authors of [48] compose their substitutions the other way, which is why they need to write their recursive 'et' rule (the last one) with the first subterms as second arguments when collecting those of the subterms.

Given two $\mathcal{FOT}$s s and t, the edit distance between them $\delta(s,t)$ is defined as follows. If the first argument is the empty string, then:

$$\delta(\varepsilon,t) \stackrel{\text{def}}{=} |t|^{\emptyset}_{|t|} \tag{G.4}$$

meaning that the edit distance is the length of the second argument, which is also this distance's maximum known normalization factor. If the first argument is a variable, then:[8]

$$\delta(X,t) \stackrel{\text{def}}{=} 0^{\{t/X\}}_{0} \tag{G.5}$$

meaning that it is zero, while binding its first argument to its second argument, with a zero normalization factor. If the second argument is a variable, then:

$$\delta(t,X) \stackrel{\text{def}}{=} \delta(X,t) \tag{G.6}$$

by symmetry, which then uses the previous case. Otherwise (neither argument is a variable), let $s = g(s_1,\ldots,s_m)$ and $t = f(t_1,\ldots,t_n)$ for some $m,n \geq 0$; then:

$$\left.\begin{aligned} \delta(s,t) \stackrel{\text{def}}{=}\ & \delta(f,g)^{\emptyset}_{\max(|f|,|g|)} \\ & + \min\{\ \delta(\varepsilon,s_1) + \delta(\varepsilon(s_2,\ldots,s_m),\ \varepsilon(t_1,\ldots,t_n)) \\ & \qquad ,\ \delta(\varepsilon,t_1) + \delta(\varepsilon(s_1,\ldots,s_m),\ \varepsilon(t_2,\ldots,t_n)) \\ & \qquad ,\ \delta(s_1,t_1) + \delta(\varepsilon(s_2,\ldots,s_m)\sigma,\varepsilon(t_2,\ldots,t_n)\sigma)\ \} \\ & \text{if } \delta(s_1,t_1) = p^{\sigma}_{q} \text{ for some } p,q \geq 0 \text{ and substitution } \sigma \end{aligned}\right\} \tag{G.7}$$

which defines a Levenshtein distance extended from strings to terms. It is equal to the edit distance between the functors plus the minimum of the three possible ways of aligning the respective sequences of subterms, composing substitutions and adding normalization factors, each set to the maximum functor length, while incrementally instantiating subterms remaining in the tails with the accumulated substitutions resulting from computing the heads' edit distance at each recursive calls.

> ***HAK's comment:*** *The above rules given as Equations (G.4)–(G.7), with my own—and (IMO) simpler—notation, are adapted from Definition 5 of the Gilbert-Schroeder paper [48]. However, while they agree on the first three rules, they do not on the last one. On that last one, they only agree on the first two cases of the three recursive patterns, they differ on the last: our own rule—Equation (G.7)—propagates to the rest of the arguments the substitution resulting from computing the edit distances between the first arguments of both terms. The two other cases need not do so as either term's first argument is only paired with the empty string ε, which simply returns the identity substitution $\emptyset$—by Equation (G.4). Indeed, not propagating like in their definition of the term edit distance 'et' (Definition 5) is incorrect. Take for instance the two terms $f(a,b)$ and $g(X,X)$. According to that definition, their term edit distance is $1^{\{a/X\}}_{3}$. However, taking that substitution into account, it should be $2^{\{a/X\}}_{3}$ (since there are 2 mismatches between $f(a,b)$ and either $g(a,a)$ or $g(b,b)$: $f \neq g$ and $a \neq b$). Indeed, the definition given in [48] means that the two occurrences of X are seen as two independent variables which then get independently bound (one time to a and the other time to b), then composing the substitutions will keep only the first one ($\{a/X\}$) and not account for the*

[8] Recall that we use Prolog's convention of writing variables with capitalized symbols.

argument mismatch $a \neq b$. Whereas, propagating the substitution as done in Equation (G.7) makes it possible to account for the mismatch (since a will be have been substituted for X), therefore correctly returning $2_3^{\{a/X\}}$.

It could have been a typo or misprint in Definition 5 in [48], perhaps. But in other later papers using this unification (such as [94] and [95]), the same definition is again given. At any rate, to propagate subsitutions from one matching argument to matching the rest is a simple option. Not doing it, although not optimal, does not invalidate their approach when the substitution propagation is done correctly (as done in Equation (G.7)); it just catches less mismatches in general than ought to be reported (since it has for effect to ignore potential mismatches that may come from any multiple-occurrence variable which are already bound to different symbols).

In this manner, this accounts for the fact it may be necessary to perform variable substitutions while establishing the normalized edit distance between two terms t_1 and t_2 such that the following fuzzy equation holds whenever $\delta(t_1, t_2) = m_n^\sigma$:

$$t_1\sigma \sim t_2\sigma \left[\tfrac{n-m}{n}\right]. \tag{G.8}$$

Indeed, having m character mismatches over a maximum total length of n characters means that the rate of mismatch is $\frac{m}{n} \in [0, 1]$; or, equivalently, that the rate of correctly matched characters is $1 - \frac{m}{n}$; *i.e.*, $\frac{n-m}{n}$. It can then be used as a truth value fuzzifying the equation $t_1\sigma = t_2\sigma$. Indeed, if all the characters are mismatched, then $m = n$ and therefore the truth value of this equation is zero; whereas, if no characters are mismatched, then $m = 0$, and the truth value is one. As expected, the higher the number of mismatches, the fuzzier the solution.

This is a very interesting trick: "*stringifying*" the syntax of a $\mathcal{FOT}$ and then using fuzzy symbol matching while counting how many mismatches over how long symbols and substituting terms for variables as needed to resolve discrepancies in term structure. Thus, calculating the normalized edit distance between two terms with Equations (G.4)–(G.7) operates an implicit unification procedure (which we shall call Gilbert-Schroeder fuzzy unification). It has, in fact, the same recursive pattern as Robinson's procedural unification algorithm, relaxed to tolerate functor and arity inequalities [87].[9]

There are some important observations to be made at this point regarding Gilbert-Schroeder fuzzy $\mathcal{FOT}$ unification.

- It applies to conventional (crisp) Prolog terms: there is no need for "fuzzy $\mathcal{FOT}$s" whatever such may be (it is the unification that is fuzzy, not the terms).
- It is a purely lexical process: it relates strings as character sequences regardless of word meaning and/or context.
- It can always derive a minimal edit distance between two terms, however unrelated they may be—the more lexically unrelated, the larger this distance will be, although it will always be finite as it is bounded by a function of the size of the terms,[10] as well as the lengths of the functor symbols in them and the number of variable re-occurrences at leaves. Normalizing with respect to the length of concatenation of the longest of each pairs of symbols appearing in corresponding subterms gives a bounded measure in $[0, 1]$ of the character mismatch rate, therefrom a fuzzy matching measure may be derived.

[9] *Op. cit.*, Section 5.8, Page 32.
[10] The number of functor nodes.

- When fuzzy-unifying two non-variable non-ε terms, their arities (number of subterms) may differ as each subterm of one is unified with each subterm of the other, keeping only the minimal total number of mismatches (and collateral substitution and normalization factor)—which raises efficiency concerns. Such concerns have been addressed for tree edit distances in more recent works such as, *e.g.*, [44], although not for $\mathcal{FOT}$s which are rooted directed acyclic graphs (variables are shared nodes). Although the number of arguments of two fuzzy matching terms may differ, it must be noted that in computing edit-distance between two $\mathcal{FOT}$s, order argument-position is always preserved.

- It is not difficult to understand from Equations (G.1) and (G.7), that the complexity of a *naïve* implementation of this recursive scheme becomes quickly prohibitive for pragmatics. Thus, optimization methods, implementation techniques such as Dynamic Programming including specific-domain heuristics, have been the center of attention [32], [104], [103].

- More worrisome is that computing this term edit distance is not only expensive, it is also *non-deterministic*. Indeed, there may be equal minimal number of mismatches with different and incomparable variable substitutions. For example, the minimal term edit distance between `f(X,X,a)` and `g(b,Y,Y)` is 2 with either substitutions $\{\texttt{X/b}, \texttt{Y/b}\}$ or $\{\texttt{X/a}, \texttt{Y/a}\}$, which are both most general although mutually incomparable (*i.e.*, they are not alphabetical variants "up to variable renaming" of one another).