L2/12-174

Simplification of Contraction Processing in UCA

Author: Richard Wordingham
Date: 4 May 2012

Markus Scherer has noted that the handling of contractions for the Unicode Collation Algorithm is surprisingly hard to code up correctly, and a solution is proposed in L2/12-131. There are two key parts.

The first is to require 'prefix contractions', which appears reasonable. It certainly simplifies coding, though it might not be unreasonable to allow prefixes to be generated on the fly in implementations.

The second part is the abandonment of contractions for 'skipped' characters. However, I am not persuaded that any great simplification actually arises. Below, I present pseudo-code for the two schemes that would result if, for one, the full proposal was accepted and for the other, if only the requirement on prefix contractions were implemented. There is very little difference in complexity. It is possible that the pseudo-code I present contains errors, but I believe it adequately captures the complexity of the algorithm.

Now it may be argued that by storing an unlimited array of booleans (labelled *M*) to be accessed by adjusted index rather than an array of characters, I have unnaturally increased complexity. However, I do not believe the difference is significant.

Variables

`T`	String for which collation weights are wanted
`N`	Length of T, in characters
`W`	Collation weights
`w`	Collation weights returned in a single look-up
`S`	String looked up in collation table
`G`	String successfully looked up in collation table. S and G could reasonably share the same storage, with the difference residing in the length.
`M`	M[i] is whether element *i* of M[i] has been used in deriving the weights. It is assumed to be expensive to maintain, and therefore only a short length of it actually exists in storage, and most of the time none if it will actually exist. Its existence is indicated by either *b* or *k* (depending on scheme) being non zero. The first relevant element of M is *k*. The cost of extending it should it be necessary is implicit in assignment to its elements. If the algorithm consumed *T* and *T* were a UTF-32 string, M[i] could be implemented by flipping the sign of elements of *T*, in which case deletion and creation logic would not be required. M[i] only matters if T[i] has non-zero combining class.
`i`	Start point of run of characters under consideration. It may be advanced within the loop of which it appears to be the loop variable.
`j`	Index of latest character being considered.

`c`	Canonical combining class of T[j].
`b`	If non-zero, canonical combining class of blocked from being added to contraction.
`k`	Index of first skipped element of T.
`r`	Highest indexed character skipped.

Functions

`cc`	Returns canonical combining class of its argument
`CE`	Returns the collation elements for its argument

Pseudo-code

Scheme Proposed in L2/12-131	**Well-Formed Constraint on Contractions Assumed; Otherwise Meets Current Definition**
`r = 0;` // Highest indexed character skipped	`k = 0;` // Make defined `r = 0;` // Highest indexed character skipped
`W = [];` // Empty table of weights for string	`W = [];`
`for (i = 0; i < N; i++) {`	`for (i = 0; i < N; i++) {`

	`    if (k) {`
	`        if (i > r) {`
	`            delete M;`
	`            k = 0;`
	`        } else if (M[i]) {`
	`            continue;`
	`         }`
	`    }`

`    S = G = T[i];`
`    w = CE(S);`
`    b = 0; //` CC of latest skipped character
	`    S = G = T[i];`
	`    w = CE(S);`
	`    b = 0; //` CC of latest skipped character

`    for (j = i+1; j < N; j++) {`	`    for (j = i+1; j < N; j++) {`
	`         //` Don't include twice `        if (k && M[j]) continue; //`

`        c = cc(T[j]);`
`        if (b > 0) {`
`            if (c == 0) break;`
`            if (c == b) {`
`                M[j] = 0;`
`                if (j > r) r = j;`
`                continue;`
`            }`
`        }`
`        S = G + T[j];`
`        if (S` has a contraction`) {`
`            G = S;`
`            w = CE(S);`
	`        c = cc(T[j]);`
	`        if (b > 0) {`
	`            if (c == 0) break;`
	`            if (c == b) {`
	`                M[j] = 0;`
	`                if (j > r) r = j;`
	`                continue;`
	`            }`
	`        }`
	`        S = G + T[j];`
	`        if (S` has a contraction`) {`
	`            G = S;`
	`            w = CE(S);`

`            if (b > 0){`
`                M[j] = 1;`
	`            if (b > 0) {`
	`               M[j] = 1;`

Scheme Proposed in L2/12-131	**Well-Formed Constraint on Contractions Assumed; Otherwise Meets Current Definition**

`            }`
`            i = j;`
	`            } else {`
	`               i = j;`
	`            }`

`        } else if (0 == c) {`	`        } else if (0 == c) {`
	`            if (k) M[i] = 0;`

`            break;`
`        } else {`
	`            break;`
	`        } else {`

`            if (0 == b) {`	`            if (0 == k) {`

`                k = j;`
`                new M;`
`            }`
`            b = c;`
`            M[j] = 0;`
`            if (j > r) r = j;`
`        }`
`    }`
`    W += w;`
	`                k = j;`
	`                new M;`
	`            }`
	`            b = c;`
	`            M[j] = 0;`
	`            if (j > r) r = j;`
	`        }`
	`    }`
	`    W += w;`

// Simple treatment of skipped points
`     if (b > 0) {`
`         for (j = k; j < i; j++) {`
`             if (!M[j]) W += CE(T[j]);`
`         }`
`         delete M;`
`     }`	

`}`	`}`