Implementations of Bidirectional Reordering Algorithms

L2/01-218

Steven Atkin
IBM, Austin, TX
atkin@us.ibm.com

Ryan Stansifer
Florida Tech, Melbourne, FL
ryan@cs.fit.edu

Abstract

The goal of this paper is to contribute to a deeper understanding of the Unicode Bidirectional Reference Algorithm. We have provided an alternative reference algorithm written in the functional language Haskell. The advantage of Haskell is that it allows for a short, clear description of a complex problem.

We have run our algorithm, the two Unicode reference implementations, and four others (ICU, PGBA, FriBidi, JDK 1.2) to test for compliance with the published standard. Conclusions are difficult to reach, but problems were found in the implementations and descriptions and above all with a character-stream to character-stream interpretation of the display of bidirectional text.

Keywords
Bidirectional layout, Arabic language processing, Hebrew language processing, Unicode

I. INTRODUCTION

Prior to the introduction of rich encoding schemes such as Unicode and ISO10646, most text streams consisted of characters originating from a single script. Traditionally an encoding was comprised of one national script plus a subset of the Latin script (ASCII 7) which fit within the confines of an 8 bit character type. In such an environment presentation of text is a trivial matter. For the most part the order in which a program stores its characters (logical order) is equivalent to the order in which they are visually presented (display order). The only exceptions are scripts written from right to left (Arabic, Hebrew, Farsi, Urdu, and Yiddish). Requiring users to enter characters in display order can solve this problem easily enough. So if a text stream contained Arabic characters the user would then simply enter them backwards. This solution, albeit not elegant, becomes cumbersome when scripts are intermixed.[3], [7], [13]

Another potential solution is to allow users to enter text in logical order but expect them to use some explicit formatting codes (for example, 0x202B and 0x202A in Unicode) for segments of text that run contrary to the base text direction. Once again this sounds acceptable, but yet causes problems. Namely what does one do with the explicit control codes in tasks other than display? For example, what effect should these controls have on searching and data interchange. These explicit codes require specific code points to be set-aside for them as well. In some encodings this may be unacceptable due to the fixed number of code points available and the number of code points required to represent the script itself. [3], [7]

Ideally one would like to maintain the flexibility of entering characters in logical order while still achieving the correct visual appearance. Fortunately such algorithms do exist and are called implicit layout algorithms. They require no explicit directional codes nor any higher order protocols. These algorithms can automatically determine the correct visual layout by simply examining the logical text stream. Yet in certain cases correct layout of a text stream may still remain ambiguous. Consider the following example in Figure 1 in which Arabic letters are represented by upper case Latin characters.

Figure 1: Ambiguous layout

fred does not believe TAHT YAS SYAWLA I

In the absence of context (a base or paragraph direction) there are two possible ways to read the sentence. When read from left to right (Fred does not believe I always say that), and when read from right to left (I always say that Fred does not believe.) [7]

The Unicode Bidirectional Algorithm rectifies such problems by providing a mechanism for unambiguously determining the visual representation of all raw streams of Unicode text. The algorithm is based upon existing implicit layout algorithms and is supplemented by the addition of explicit directional control codes. Generally the implicit rules are sufficient for the layout of most text streams. Then again there are cases in which the algorithm may give inappropriate results. Consider a phone number appearing in a stream of Arabic letters, MY NUMBER IS (321)713-0261. This should not be rendered as a mathematical expression [3],[13]. See Figure 2.

Figure 2: Rendering numbers

0261-713(321) SI REBMUN YM **(incorrect)**

(321)713-0261 SI REBMUN YM **(correct)**

Without understanding the use of the numbers in this context the correct display cannot be determined. There are numerous contextually and cultural factors in determining the display order. These situations could be overcome through the use of explicit directional controls.

This paper discusses the results of various implementations of the Unicode Bidirectional Algorithm as published in Unicode Technical Report #9 [14] as well as offering a purely functional description of the algorithm for implementers. Specifically the following implementations will be examined Pretty Good Bidi Algorithm (PGBA) [10], Free Implementation of the Bidi Algorithm (FriBidi) [5], IBM Classes for Unicode (ICU) [8], Java 1.2 [4], Unicode Java Reference [14], Unicode C Reference [14], and our own Haskell Bidi (HaBi).

II. Reference Implementation

Currently there exist two reference implementations of the Unicode Bidirectional algorithm: Java and C as well as a printed textual description (Unicode Technical Report #9) [14]. One might ask why implement the Unicode Bidirectional algorithm in a purely functional language when so many other implementations already exist? It is the authors contention that a greater understanding of the algorithm is best obtained by a clear functional description of its operations [6]. Without a clear description implementers may encounter ambiguities that ultimately lead to divergent implementations, contrary to the primary goal of the Unicode Bidirectional Algorithm. Lastly we were interested in determining if the algorithm could be implemented without an examination of the Java implementation on the Unicode 3.0 CD.

III. Hugs 98 Implementation

In this section the source code to HaBi is presented. The HaBi reference implementation uses the Hugs 98 version of Haskell 98 [9] as it is widely available (Linux, Windows, and Macintosh) and easily configurable.

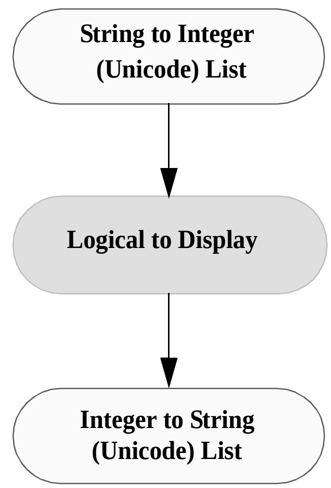
Since the dominant concern in HaBi is comprehension and readability our implementation closely follows the textual description as published in the Unicode Technical Report #9. See Figure 4. HaBi is comprised of five phases as in the Java Unicode Bidirectional Reference implementation:

- Resolution of explicit directional controls
- Resolution of weak types
- Resolution of neutral types
- Resolution of implicit levels
- Reordering of levels

Currently there is no direct support for Unicode in the Hugs 98 implementation of Haskell 98[1]. So we treat Unicode as lists of 16 or 32 bit integers. The authors provide two modules for Unicode manipulation. The first is used to create Unicode (UCS4, UCS2, and UTF-8) strings. The second is used for determining character types. Utility functions convert Haskell strings with optional Unicode character escapes to 16 or 32 bit integer lists. A Unicode escape takes the form \u*hhhh* analogous to Java. This escape sequence is used for representing code points outside the range 0x00 - 0x7f. This format was chosen so as to permit easy comparison of results to other implementations.

Internally HaBi manipulates Unicode as sequences of 32 bit integers. See Figure 3. HaBi is prepared to handle surrogates as soon as Unicode assigns them. The only change HaBi requires is an updated character attribute table. It would be more elegant to use the polymorphism of Haskell since the algorithm does not really care about the type of a character only its attribute.

Figure 3: Input and output of Haskell Bidirectional Reference



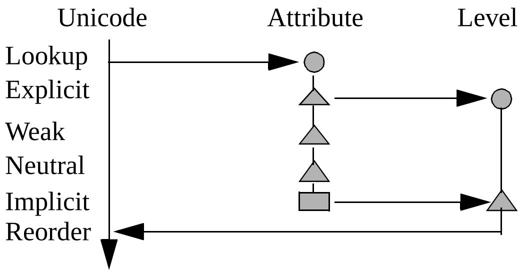
Each Unicode character has an associated Bidirectional attribute and level number. Figure 4 shows the general relationship of this information throughout the steps of the algorithm. The first step in our implementation is to lookup and assign bidirectional attributes to the logical character stream. The attributes are obtained from the online character database as published in Unicode 3.0. At this point explicit processing assigns level numbers as well as honoring any directional overrides. Weak and neutral processing potentially causes attribute types to change based upon surrounding attribute types. Implicit processing assigns final level numbers to the stream which control reordering. Reordering then produces a sequence of Unicode characters in display order.

Figure 4: Data flow

HaBi uses the following three internal types:

- type Attributed = (Ucs4, Bidi)
- type Level= (Int, Ucs4, Bidi)
- data Run = LL[Level] | LR[Level] | RR[Level] | RL[Level]

1. The Haskell 98 Report defines the Char type as an enumeration consisting of 16 bit values conforming to the Unicode standard. The escape sequence used is consistent with that of Java (\u*hhhh*). Unicode is permitted in any identifier or any other place in a program. Currently the only Haskell implementation known to support Unicode directly is the Chalmers' Haskell Interpreter and Compiler.



Wherever possible the implementation treats characters collectively as sequential runs rather than as individual characters [1]. By using one of data type Run's four possible type constructors characters can then be grouped by level. These four constructors signify the possible combinations of starting and ending run directions. For example, the LL constructor signifies that the start of a run and the end of a run are both left to right. Therefore runs of LL followed by RL are not created.

Before the details of the source code are discussed it is important to make note of the following concerning HaBi:

- The logical text stream is assumed to have already been separated into paragraphs and lines.
- Directional control codes are removed once processed.
- No limit is imposed on the number of allowable embeddings.
- Mirroring is accomplished by performing character replacement.

By separating those facets of layout dealing with reordering from those that are concerned with rendering (line breaking, glyph selection, and shaping) comphrension of the Haskell implementation is more discernible. In the source code functions are named in such a way so as to correspond to the appropriate section in the Unicode Bidirectional textual reference [14]. See Appendix. For example, the function named weak refers to overall weak resolution. While the function named w1_7 lines 45-71 specifically refers to steps 1 through 7 in weak resolution.

The function logicalToDisplay lines 150-158 in the Appendix, is used to convert a stream in logical order to one in display order. First, calls to the functions explicit lines 37-41, weak lines 73-78, neutral lines 94-99, and implicit lines 114-119 form runs of fully resolved characters. Calls to reorder lines 134-140 and mirror lines 142-148 are then applied to the fully resolved runs which in turn yield a stream in display order. This is discussed in greater detail in the next few paragraphs.

The explicit function breaks the logical text stream into logical runs via calls to p2_3 lines1-8, x2_9 lines 10-27 and x10 lines 29-35. The reference description suggests the use of stacks for keeping track of levels, overrides, and embeddings. In our implementation stacks are used as well, but they are implicit rather than explicit (function x2_9 arguments two, three, and four). The functions weak, neutral, and implicit are then mapped onto each individual run.

In weak steps 1 though 7 lines 45-71 two pieces of information are carried forward (the second and third arguments of function w1_7) the current directional state and the last character's type. There are cases in the algorithm where a character's direction gets changed but the character's intrinsic type remains unchanged. For example, if a stream contained an AL followed by a EN the AL would change to type R (step three in weak types resolution). However the last character would need to remain AL so as to cause the EN to change to AN (step two in resolution of weak types).

The functions n1_2 lines 80-92 and i1_2 lines 102-112 resolve the neutral and implicit character types respectively. The details of these functions are not discussed as they are fairly straight forward. At this point runs are fully resolved and ready for reordering (function reorder).

Reordering occurs in two stages. In the first stage (function reverseRun lines 121-126), a run is either completely reversed or left as is. This decision is based upon whether a run's level is even or odd. If it is odd (right to left) then it is reversed. In the second stage (function reverseLevels lines 128-132), the list of runs are reordered. At first it may not be obvious that the list being folded is not the list of runs, but is the list of levels (highest level to the lowest odd level in the stream). Once reordering is finished the list of runs are collapsed into a single list of characters in display order.

IV. Alternative Implementations

The layout of bidirectional text is a complex endeavour with no single right answer. There is limited agreement about which tasks should be part of a bidirectional algorithm. To simply compare the features of each implementation as if one were buying a refrigerator would be unfair.

There are several places in the text of the Unicode Bidirectional Algorithm that make reference to features that most certainly need to be part of a complete bidirectional layout solution but do not necessarily need to be part of a reordering algorithm.

In Arabic, and to some extent in Hebrew, the mapping of a glyph (visual shape) to a character is not one-to-one as in the Latin script. Instead the selection of a character's glyph is based upon its position within a word. Each Arabic character may have up to four possible shapes: [2], [4], [12]

- Initial -Character appears in the beginning of a word
- Final - Character appears at the end of a word.
- Medial - Character appears somewhere in the middle.
- Isolated - Character is surrounded by white space.

Furthermore, glyph selection must also take into consideration the linking abilities of the surrounding characters. For example, some glyphs may only link on their right side while others may permit links on either side. In Arabic each character belongs to one of the following joining classes: [13]

- Right joining - Alef, Dal, Thal, Zain
- Left joining - None
- Dual joining - Beh, Teh, Theh, ...
- Join causing - Tatweel, Joiner (U200D)
- Non joining - Spacing characters, Non-joiner (U200C)
- Transparent -Combining marks

In Hebrew some characters do have final forms even though Hebrew is not a cursive script. The idea of contextual shaping is certainly not limited just to right to left scripts. For example, the Greek script provides a special final form for the sigma character. [13]

Occasionally two or more glyphs combine to form a new single glyph called a ligature. This resultant shape then replaces the individual glyphs from which it is comprised. In Arabic this occurs frequently and in Hebrew rarely. In particular the Alef Lamed ligature (UFB4F) is used in liturgical books. Although infrequent, ligatures do occur even in English. Specifically, the ﬁ ligature where the letter f merges with the letter i. [2], [4], [12]

In some cases glyph selection may be based on a character's resolved direction. These characters are known as mirrored characters (parentheses and brackets). When mirrored characters are intermixed with Arabic and or Hebrew characters, a complementary shape may need to be selected so as to preserve the correct meaning of an expression. For example, consider the text stream 1 < 2 in logical order (one less than two). If this stream is to be displayed in a right to left sequence it must be displayed as 2 > 1. In order to preserve the correct meaning the < is changed to >. This process is known as mirroring or symmetric swapping. [12]

Traditionally glyphs are selected and drawn by font rendering engines rather than via character replacement. The reasons for this approach is centered around glyph availability. Some glyphs may simply not be available in a font (Hebrew and Greek final forms). If implementers were to replace sequences of characters with new character ligatures, there would be no guarantee that they would be present in a font as well. Some ligatures are not able to be constructed by using character replacement, as they are not present in Unicode. The choice of an appropriate glyph requires knowledge of the font and its available glyphs.

Also the process of line breaking requires font knowledge. Line breaking determines where a line begins and ends within some graphical object. It is simply not possible to do this effectively without knowing the widths of all the glyphs along with the width of the display area. Implementers can not assume that the number of and width of each character is the same when rendered. [2], [4]

Putting aside glyph related problems there are still other facets in a complete layout solution. For example, it may not always be possible to determine where a paragraph begins and ends by examining just the stream contents. Higher order information is required. This information could appear as control codes or be supplied externally. [13]

Higher order external information can also force the stream contents to change. Most countries use European numerals for representing digits. Some countries use another form, Arabic numerals along with the European numerals. Higher order data may override a particular representation. [7]

There are also aspects of bidirectional layout that are outside the scope of higher order protocols. In particular the caret and the mouse. Movement of the caret and hit testing of the mouse becomes more complex in bidirectional streams. If the caret is moving linearly within one of the streams (logical or visual) then this movement needs to be translated to the other stream. Highlighting poses a similar problem as to which stream is being highlighted (logical or visual). [2], [4]

Unfortunately the tasks that one would like to provide are not necessarily the same ones that can be provided. All of this depends upon the intended use of an implementation. If the intended use is to fit within in some broader context then it may be acceptable to leave some features out. If the intended use is to provide a complete internationalization framework a set of features above and beyond the ones mentioned may be required. The specification of the bidirectional algorithm can only be implemented as a character stream reordering (What else can an implementer do?), yet the bidirectional layout problem can only be solved in a larger context. However, we will summarize the features of each model for those who like to make comparisons. See Table 1.

Java 1.2 provides a complete framework for creating multi script applications. Java's TextLayout and LineBreakMeasurer classes facilitate the layout of complex text in a platform neutral manner. The underlying approach to reordering is based on the Unicode Bidirectional Algorithm. [4]

ICU's approach is very close to Java due in some respect to the fact that the overall internationalization architecture of Java is based on ICU. The key differences are centered around glyph management. The lack of glyph management routines should not be interpreted as a deficiency but rather as a statement as to the context in which ICU is to be used. [8]

Mark Leisher's PGBA is another algorithm for bidirectional reordering. The algorithm takes an implicit approach to reordering. PGBA does not attempt to match Unicode's reordering algorithm. However PGBA's implicit algorithm does match the implicit section of the Unicode Bidirectional Algorithm. At the moment it does not support the explicit bidirectional control codes (LRE, LRO, RLE, RLO, PDF). One should not infer that the lack of support for directional control codes results in an incomplete algorithm. Under most circumstances the implicit algorithm reorders a text stream correctly. Secondly, these control codes are not always present in all encoding schemes. Of course it would be a nice feature, but certainly not a necessary one. [10]

Dov Grobgeld's FriBidi follows the Unicode Bidirectional Reference more closely. Notably there is support for integration with graphical user interfaces along with a collection of codepage converters. However as in PGBA the explicit control codes are not currently supported. [5]

V. Evaluation Methodology

The primary goal in evaluating the algorithms was to garner whether or not their output matched Unicode's reference algorithm. We have tested them on a large number of small, carefully crafted test cases of basic bidirectional text. To simulate Arabic and Hebrew input/output a simple set of rules are utilized. These rules make use of characters from the Latin-1 charset. The character mappings allow Latin-1 text to be used instead of real Unicode characters for Arabic, Hebrew, and

Table 1: Feature summary

Feature	Java 1.2.2	ICU 1.5	PGBA 2.4	FriBidi 1.12	HaBi 1.0
Reordering	•	•	•	•	•
Shaping	•				
Mirroring	•	•	•	•	•
Drawing	•				
Caret	•	•	•	•	
Hit testing	•			•	
Highlighting	•			•	
Line break	•				
Bounding box	•			•	
Font	•				

control codes. This is an enormous convenience in writing, reading, running and printing the test cases. This form is the same as the one used by the Unicode Bidirectional Reference Java Implementation [14]. See Table 2. Unfortunately not all of the implementations adhere to these rules in their test cases. To compensate for this, changes were made to some of the implementations.

Table 2: Bidirectional character mappings

Type	Arabic	Hebrew	Mixed	English
L	a - z	a - z	a - z	a - z
AL	A - Z		A - M	
R		A - Z	N - Z	
AN	0 - 9		5 - 9	
EN		0 - 9	0 - 4	0 - 9
LRE	[	[	[	[
LRO	{	{	{	{
RLE	]	]	]	]
RLO	}	}	}	}
PDF	^	^	^	^
NSM	~	~	~	~

In the Unicode C reference implementation additional character mapping tables were added to match those of the Unicode Java Reference implementation. Also the bidirectional control codes were remapped from the control range 0x00-0x1F to the printable range 0x20-0x7E. This remapping allows test results to be compared more easily.

In PGBA and FriBidi the character attribute tables were modified to match the character mappings outlined in Table 2. However, the strategy for testing ICU and Java was slightly different. In the ICU and Java test cases the character types are used rather than a character mapping. So in places where our test cases required a specific type, that type was simply used rather than a character mapping.

The test cases are presented in Tables 3-6. The source column of each table shows the test input. The expected column is what we think the correct output should be. In all cases this is the output produced by our HaBi implementation. These test cases are taken from the following sources:

- Mark Leisher - His web page provides an excellent suite of test cases as well as a table of results for other implementations [10]. See Tables 3-4.
- Unicode Technical Report #9 - Some of the examples are used for testing conformance [14]. See Table 3.
- Additional test cases for uncovering potential bugs in an implementation's handling of weak types and directional controls. See Tables 5-6.

Table 3: Arabic charmap tests

	Source	Expected
1	car is THE CAR in arabic	car is RAC EHT in arabic
2	CAR IS the car IN ENGLISH	HSILGNE NI the car SI RAC
3	he said “IT IS 123, 456, OK”	he said “KO ,456 ,123 SI TI”
4	he said “IT IS (123, 456), OK”	he said “KO ,(456 ,123) SI TI”
5	he said “IT IS 123,456, OK”	he said “KO ,123,456 SI TI”
6	he said “IT IS (123,456), OK”	he said “KO ,(123,456) SI TI”
7	HE SAID “it is 123, 456, ok”	“it is 123, 456, ok” DIAS EH
8	<H123>shalom</H123>	<123H/>shalom<123H>
9	HE SAID “it is a car!” AND RAN	NAR DNA “!it is a car” DIAS EH
10	HE SAID “it is a car!x” AND RAN	NAR DNA “it is a car!x” DIAS EH
11	-2 CELSIUS IS COLD	DLOC SI SUISLEC -2
12	SOLVE 1*5 1-5 1/5 1+5	5+1 5/1 5-1 5*1 EVLOS
13	THE RANGE IS 2.5..5	5..2.5 SI EGNAR EHT
14	IOU $10	10$ UOI
15	CHANGE -10%	%10- EGNAHC

Table 3: Arabic charmap tests (Continued)

	Source	Expected
16	-10% CHANGE	EGNAHC %10-
17	he said “IT IS A CAR!”	he said “RAC A SI TI!”
18	he said “IT IS A CAR!X”	he said “X!RAC A SI TI”
19	(TEST) abc	abc (TSET)
20	abc (TEST)	abc (TSET)
21	#@$ TEST	TSET $@#
22	TEST 23 ONCE abc	abc ECNO 23 TSET
23	he said “THE VALUES ARE 123, 456, 789, OK”	he said “KO ,789 ,456 ,123 ERA SEULAV EHT”.
24	he said “IT IS A bmw 500, OK.”	he said “A SI TI bmw KO ,500.”

Table 4: Hebrew charmap tests

	Source	Expected
1	HE SAID “it is 123, 456, ok”.	.”it is 123, 456, ok” DIAS EH
2	<H123>shalom</H123>	<123H/>shalom<123H>
3	<h123>SAALAM</h123>	<h123>MALAAS</h123>
4	-2 CELSIUS IS COLD	DLOC SI SUISLEC -2
5	-10% CHANGE	EGNAHC -10%
6	TEST ~~~23%%% ONCE abc	abc ECNO 23%%%~~~ TSET
7	TEST abc ~~~23%%% ONCE abc	abc ECNO abc ~~~23%%% TSET
8	TEST abc@23@cde ONCE	ECNO abc@23@cde TSET
9	TEST abc 23 cde ONCE	ECNO abc 23 cde TSET
10	TEST abc 23 ONCE cde	cde ECNO abc 23 TSET
11	Xa 2 Z	Z a 2X

Table 5: Mixed charmap tests

	Source	Expected
1	A~~	~~A
2	A~a~	a~~A
3	A1	1A
4	A 1	1 A
5	A~1	1~A
6	1	1
7	a 1	a 1
8	N 1	1 N
9	A~~ 1	1 ~~A
10	A~a1	a1~A
11	N1	1N
12	a1	a1
13	A~N1	1N~A
14	NOa1	a1ON
15	1/2	1/2
16	1,2	1,2
17	5,6	5,6
18	A1/2	2/1A
19	A1,5	1,5A
20	A1,2	1,2A
21	1,.2	1,.2
22	1,A2	2A,1
23	A5,1	5,1A
24	+$1	+$1
25	1+$	1+$
26	5+1	5+1
27	A+$1	1$+A

Table 5: Mixed charmap tests (Continued)

	Source	Expected
28	A1+$	$+1A
29	1+/2	1+/2
30	5+	5+
31	+$	+$
32	N+$1	+$1N
33	+12$	+12$
34	a/1	a/1
35	1,5	1,5
36	+5	+5

Table 6: Explicit override tests

	Source	Expected
1	a}}}def	afed
2	a}}}DEF	aFED
3	a}}}defDEF	aFEDfed
4	a}}}DEFdef	afedFED
5	a{{{def	adef
6	a{{{DEF	aDEF
7	a{{{defDEF	adefDEF
8	a{{{DEFdef	aDEFdef
9	A}}}def	fedA
10	A}}}DEF	FEDA
11	A}}}defDEF	FEDfedA
12	A}}}DEFdef	fedFEDA
13	A{{{def	defA
14	A{{{DEF	DEFA
15	A{{{defDEF	defDEFA
16	A{{{DEFdef	DEFdefA
17	^^abc	abc
18	^^}abc	cba
19	}^abc	abc
20	^}^abc	abc
21	}^}abc	cba
22	}^{abc	abc
23	}^^}abc	cba
24	}}abcDEF	FEDcba

Table 7: Arabic test differences

	PGBA 2.4	FriBidi 1.12	Unicode C Reference
2		SI RAC the car NI ENGLISH	
4	he said “KO ,)456 ,123(SI TI”		
6	he said “KO ,)123,456(SI TI”		
7		DIAS EH “it is 456 ,123, ok”	“ok ,456 ,123 it is” DIAS EH
8		<123H>shalom</123H>	
9		DIAS EH “it is a car!” DNA RAN	
10		DIAS EH “it is a car!x” DNA RAN	
11		-SI SUISLEC 2 COLD	DLOC SI SUISLEC 2-
12	1+5 1/5 1-5 5*1 EVLOS		
14	$10 UOI		
15	%-10 EGNAHC	10- EGNAHC%	
16	EGNAHC %-10	-10% CHANGE	
19	abc)TSET(	(TSET) abc	
21		#@$ TEST	
22		ECNO 23 TSET abc	
24	he said “A SI TI bmw 500, KO.”		

VI. Test Results

All implementations were tested by using the test cases from Tables 3-5. The implementations that support the Unicode directional control codes (LRO, LRE, RLO, RLE, and PDF) were further tested using the test cases from Table 6. At this time the directional control codes are only supported by HaBi, ICU, Java 1.2, Unicode Java reference, and Unicode C reference.

When the results of the test cases were compared, the placement of directional control codes and choice of mirrors was ignored. This is permitted as the final placement of control codes is arbitrary and mirroring may optionally be handled by a higher order protocol.

Tables 7-9 detail the differences among the implementations with respect to the results obtained with the HaBi Implementation. Only PGBA, FriBidi and the Unicode C implementation return results that are different than the HaBi implementation. The Unicode Java reference, Java 1.2, and ICU pass all test cases.

Table 8: Hebrew test differences

	PGBA 2.4	FriBidi 1.12
5	EGNAHC %-10	
6	abc ECNO %%%23~~~ TSET	
7	abc ECNO %%%23~~~ abc TSET	
11	Z 2 aX	a 2X

Table 9: Mixed test differences

	PGBA 2.4	FriBidi 1.12
1		A~~
2	~a~A	~Aa~
10	1a~A	~Aa1
14	1aON	
18	1/2A	1/2A
19		5,1A
21		2.,1
23		1,5A
27		+$1A
28		1+$A
32	1$+N	
35		5,1

In PGBA, types AL and R are treated as being equivalent [10]. This in itself does not present a problem as long as the data stream is free of AL and EN (European number). However, a problem arises when AL is followed by a EN for example, test case 18 from Table 5. In this situation the ENs should be treated as ANs (Arabic number) and not left as ENs.

The handling of NSM is also different in PGBA. PGBA treats NSM as being equal to ON (other neutral) [10]. This delays the handling of NSM until the neutral type resolution phase rather than in the weak type resolution phase. By delaying their handling, the wrong set of rules are used to resolve the NSM type. For example, in test case 2 from Table 5 the last NSM should be treated as type L instead of type R.

In FriBidi there are a few bugs in the implementation. Specifically, when an AL is followed by a EN the EN is not being changed to type AN. See test case 18 in Table 5. This is the same symptom as was found in PGBA, but the root cause is different. In FriBidi, step W2 (weak processing phase rule two) the wrong type is being examined it should be type EN instead of type N.

There is also a bug in determining the first strong directional character. The only types that are recognized as having a strong direction are types R and L. Type AL should also be recognized as a strong directional character. For example, when test case 1 from Table 5 is examined FriBidi incorrectly determines that there are no strong directional characters present. It then proceeds to default the base direction to type L when it should actually be of type R. This problem also causes test cases 2, 9, and 11 from Table 3 to fail.

VII. Conclusions

The biggest hindrance to the creation of a mechanism for converting logical data streams to display streams lies in the problem description. The problem of bidirectional layout is ill defined with respect to the input(s) and output(s).

Certainly the most obvious input is the data stream itself. But several situations require additional input in order to correctly determine the output stream. For example, in Farsi mathematical expressions are written left to right while in Arabic they are written right to left [7]. This may require a special sub input (directional control code) to appear within the stream for proper handling to occur. If it becomes necessary to use control codes for obtaining the desired results the purpose of an algorithm becomes unclear.

The situation becomes even more cloudy when one considers other possible inputs (paragraph levels, line breaks, shaping, directional overrides, numeric overrides, etc.) Are they to be treated as separate inputs? If they are treated as being distinct, when, where and how should they be used?

Determining the output(s) is not simple either. The correct output(s) is largely based on the context in which an algorithm will be used. If an algorithm is used to render text, then appropriate outputs might be a glyph vector and a set of screen positions. On the other hand, if an algorithm is simply being used to determine character reordering, then an acceptable output might just be a reordered character stream.

The Unicode Bidirectional algorithm has gone through several iterations over the years. The current textual reference has been greatly refined. Nevertheless, we believe that there is still room for improvement. Implementing a bidirectional layout algorithm is not a trivial matter even when one restricts an implementation to just reordering. Part of the difficulty can be attributed to the textual description of the algorithm. Additionally there are areas that require further clarification.

As an example consider step L2 of the Unicode Bidirectional Reference Algorithm. It states the following, “From the highest level found in the text to the lowest odd level on each line, reverse any contiguous sequence of characters that are at that level or higher. [14]” This has more than one possible interpretation. It could mean that once the highest level has been found and processed the next level for processing should be one less than the current level. It could also be interpreted as meaning that the next level to be processed is the next lowest level actually present in the text, which may be greater than one less than the current level. It was only through an examination of Unicode’s Java implementation that we were able to determine the answer. (The next level is one less than the current.)

There are also problems concerning the bounds of the Unicode Bidirectional Algorithm. In the absence of higher order protocols it is not always possible to perform all the steps of the Unicode Bidirectional Algorithm. In particular, step L4 requires mirrored characters to be depicted by mirrored glyphs if their resolved directionality is R. However, glyph selection requires knowledge of fonts and glyph substitution tables. One possible mechanism for avoiding glyph substitutions is to perform mirroring via character substitutions. In this approach mirrored characters are replaced by their corresponding character mirrors. In most situations this approach yields the same results. The only drawback occurs when a mirrored character does not have its corresponding mirror encoded in Unicode. For example, the square root character (U221A) does not have its corresponding mirror encoded.

Such situations have placed developers in a quandary. One solution is to use the implementations (Java and C) as a reference. But these implementations don't agree in every case. Furthermore the implementations have different goals. The Java implementation follows the textual reference closely while the C implementation offers performance improvements.

We argue that if source code is now going to serve as a reference we should pick source code that is more attuned to describing algorithms. We claim to have provided such a reference through the use of Haskell 98. Our HaBi reference is clear and succinct. The total number of lines of source code for the complete solution is less than 300 lines. The Unicode Java reference implementation is over 1000 lines [14].

By using a functional language we are able to separate details that are not directly related to the algorithm. In HaBi reordering is completely independent from character encoding. It does not matter what character encoding one uses (UCS4, UCS2, or UTF8). The Haskell type system and HaBi character attribute function allows the character encoding to change while not impacting the reordering algorithm. Other implementations may find this level of separation difficult to achieve (Java and C). In C the size of types are not guaranteed to be portable, making C unsuitable as a reference. In the Java reference implementation the ramifications of moving to UCS4 are unclear. Our reference presents the steps as simple, easy to understand, functions without side effects. This allows implementers to comprehend the true meaning of each step in the algorithm independently of the others while yet remaining free from language implementation details. The creation of test cases is thus more systematic.

Even if we could separate out the appropriate inputs and outputs to a reordering algorithm, there are still other problems to address. Bidirectional algorithms have been around for some time. Do we incorporate legacy algorithms even if they don't conform to our new model? If the answer is yes, then we should consider adopting an algorithm that clearly separates reordering activities and responsibilities. This new reordering algorithm could be structured in such a way so as to allow for multi-phases, multi-protocols, and reversibility. Thus permitting detection and examination of streams that have been bidirectionally processed from those that have not.

In summary the contributions of this paper are:

- We have argued that the problem of bidirectional display is ill defined causing uncertainties for implementers.
- We have demonstrated that a functional description serves as a good algorithmic reference.
- We have collected a suite of test cases for uncovering problems in regions that are difficult to implement.
- We have identified potentially difficult areas in Unicode's reference description.

In the future we plan to provide a model for creating a bidirectional algorithm that fits within the design principles of Unicode. We expect this new model to be based upon Unicode metadata and tagging. Additionally, we believe this new framework will be useful for describing general higher order protocols within Unicode.

VIII. Further Information

The full distribution of Hugs 98 is available at:

- http://haskell.org/hugs

The full distribution of HaBi is available at:

- http://www.cs.fit.edu/~satkin/i18n.html

IX. References

[1] Abramson, Dean. "Optimized Implementations of Bidirectional Text Layout and Bidirectional Caret Movement." *Thirteenth International Unicode Conference,* September 1998.

[2] Apple Computer. *Inside Macintosh Text.* Addison-Wesley. 1993.

[3] Becker, Joseph. "Arabic Word Processing." *Communications of the ACM,* July 1987, Volume 30, Number 7, pp 600-610.

[4] Davis, Mark. et al. "International Text In JDK 1.2." Available: *http://www.ibm.com/java/education/international-text/*. Retrieved: July 17, 2000.

[5] Grobgeld, Dov. "A Free Implementation of the Unicode Bidi Algorithm." Available: *http://imagic.weizmann.ac.il/~dov/freesw/FriBidi/.* Retrieved: July 17, 2000.

[6] Hughes, John. "Why Functional Programming Matters." *Computer Journal,* 1989, Volume 32, Number 2, pp 98-107.

[7] IBM Corporation. *National Language Support Bidi Guide, NLDG Volume 3.* IBM Canada Ltd. 1995.

[8] IBM Corporation. "IBM Classes for Unicode." Available: *http://www.ibm.com/java/tools/international-classes/index.html.* Retrieved: July 17, 2000.

[9] Jones, Simon P. et al. "Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language." *Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106,* February 1999.

[10] Leisher, Mark. "The UCData Unicode Character Properties and Bidi Algorithm Package." Available: *http://crl.nmsu.edu/~mleisher/ucdata.html.* Retrieved: July 17, 2000.

[11] O'Donnell, Sandra M. *Programming for the World - A Guide to Internationalization.* Prentice Hall. 1994.

[12] Sun Microsystems. “Complex Text Layout Language Support in the Solaris Operating Environment.” Available: *http://www.sun.com/software/white-papers/wp-cttlanguage/.* Retrieved: July 17, 2000.

[13] Unicode Consortium, The. *The Unicode Standard, Version 3.0.* Addison-Wesley. 2000.

[14] Unicode Consortium, The. “Unicode Technical Report #9 - The Bidirectional Algorithm.” Available: *http://www.unicode.org/unicode/reports/tr9/tr9-6.html* Retrieved: July 17, 2000.

X. Appendix

```
-- Rule P2, P3 determine base level of text from the first strong
-- directional character
p2_3 :: [Attributed] -> Int
p2_3 [] = 0
p2_3 ((_,L):xs) = 0
p2_3 ((_,AL):xs) = 1
p2_3 ((_,R):xs) = 1
p2_3 (_:xs) = p2_3(xs)

-- Rules X2 - X9
x2_9 :: [Int] -> [Bidi] -> [Bidi] -> [Attributed] -> [Level]
x2_9 _ _ _ [] = []
x2_9 (l:ls) os es ((x,RLE):xs)
  = x2_9 ((add l R):l:ls) (N:os) (RLE:es) xs
x2_9 (l:ls) os es ((x,LRE):xs)
  = x2_9 ((add l L):l:ls) (N:os) (LRE:es) xs
x2_9 (l:ls) os es ((x,RLO):xs)
  = x2_9 ((add l R):l:ls) (R:os) (RLO:es) xs
x2_9 (l:ls) os es ((x,LRO):xs)
  = x2_9 ((add l L):l:ls) (L:os) (LRO:es) xs
x2_9 ls os (e:es) ((x,PDF):xs)
  | elem e [RLE,LRE,RLO,LRO] = x2_9 (tail ls) (tail os) es xs
x2_9 ls os es ((x,PDF):xs)
  = x2_9 ls os es xs
x2_9 ls os es ((x,y):xs)
  | (head os) == N = ((head ls),x,y) : x2_9 ls os es xs
  | otherwise = ((head ls),x,(head os)) : x2_9 ls os es xs

-- Rule X10 group characters by level
x10 :: (Int, Int) -> [Level] -> Run
x10 (sor,eor) xs
  | even sor && even eor = LL xs
  | even sor && odd eor = LR xs
  | odd sor && even eor = RL xs
  | otherwise = RR xs

-- Process explicit characters X1 - X10
explicit :: Int -> [Attributed] -> [Run]
explicit l xs = zipWith x10 (runList levels l l) groups
  where levels = (map (\x -> level (head x)) groups)
  groups = groupBy levelEql (x2_9 [l][N][] xs)



-- Rules W1 - W7
w1_7 :: [Level] -> Bidi -> Bidi -> [Level]
w1_7 [] _ _ = []
w1_7 ((x,y,L):xs) _ _ = (x,y,L):(w1_7 xs L L)
w1_7 ((x,y,R):xs) _ _ = (x,y,R):(w1_7 xs R R)
w1_7 ((x,y,AL):xs) _ _ = (x,y,R):(w1_7 xs AL R)
w1_7 ((x,y,AN):xs) dir _ = (x,y,AN):(w1_7 xs dir AN)
w1_7 ((x,y,EN):xs) AL _ = (x,y,AN):(w1_7 xs AL AN)
w1_7 ((x,y,EN):xs) L _ = (x,y,L):(w1_7 xs L EN)
w1_7 ((x,y,EN):xs) dir _ = (x,y,EN):(w1_7 xs dir EN)
w1_7 ((x,y,NSM):xs) L N   = (x,y,L):(w1_7 xs L L)
w1_7 ((x,y,NSM):xs) R N   = (x,y,R):(w1_7 xs R R)
w1_7 ((x,y,NSM):xs) dir last = (x,y,last):(w1_7 xs dir last)
w1_7 ((a,b,ES):(x,y,EN):xs) dir EN =
  (a,b,EN):(x,y,EN):(w1_7 xs dir EN)
w1_7 ((a,b,CS):(x,y,EN):xs) dir EN =
  (a,b,EN):(x,y,EN):(w1_7 xs dir EN)
w1_7 ((a,b,CS):(x,y,EN):xs) AL AN  =
  (a,b,AN):(x,y,AN):(w1_7 xs AL AN)
w1_7 ((a,b,CS):(x,y,AN):xs) dir AN  =
  (a,b,AN):(x,y,AN):(w1_7 xs dir AN)
w1_7 ((x,y,ET):xs) dir EN = (x,y,EN):(w1_7 xs dir EN)
w1_7 ((x,y,z):xs) dir last
  | z==ET && findEnd xs ET == EN && dir /= AL
    = (x,y,EN):(w1_7 xs dir EN)
  | elem z [CS,ES,ET] = (x,y,ON):(w1_7 xs dir ON)
  | otherwise = (x,y,z):(w1_7 xs dir z)

-- Process a run of weak characters W1 - W7
weak :: Run -> Run
weak (LL xs) = LL (w1_7 xs L N)
weak (LR xs) = LR (w1_7 xs L N)
weak (RL xs) = RL (w1_7 xs R N)
weak (RR xs) = RR (w1_7 xs R N)

-- Rules N1 - N2
n1_2 :: [[Level]] -> Bidi -> Bidi -> Bidi -> [Level]
n1_2 [] _ _ base = []
n1_2 (x:xs) sor eor base
  | isLeft x = x ++ (n1_2 xs L eor base)
  | isRight x  = x ++ (n1_2 xs R eor base)
  | isNeutral x && sor == R && (dir xs eor) == R
    = (map (newBidi R) x) ++ (n1_2 xs R eor base)
  | isNeutral x && sor == L && (dir xs eor) == L
    = (map (newBidi L) x) ++ (n1_2 xs L eor base)
  | isNeutral x =
    (map (newBidi base) x) ++ (n1_2 xs sor eor base)
  | otherwise = x ++ (n1_2 xs sor eor base)

-- Process a run of neutral characters N1 - N2
neutral :: Run -> Run
neutral (LL xs) = LL (n1_2 (groupBy neutralEql xs) L L L)
neutral (LR xs) = LR (n1_2 (groupBy neutralEql xs) L R L)
neutral (RL xs) = RL (n1_2 (groupBy neutralEql xs) R L R)
neutral (RR xs) = RR (n1_2 (groupBy neutralEql xs) R R R)
```

```
-- Rule I1, I2
i1_2 :: [[Level]] -> Bidi -> [Level]
i1_2 [] _ = []
i1_2 ((x:xs):ys) dir
  | attrib x == R && dir == L
    = (map (newLevel 1) (x:xs)) ++ (i1_2 ys L)
  | elem (attrib x) [AN,EN] && dir == L
    = (map (newLevel 2) (x:xs)) ++ (i1_2 ys L)
  | elem (attrib x) [L,AN,EN] && dir == R
    = (map (newLevel 1) (x:xs)) ++ (i1_2 ys R)
i1_2 (x:xs) dir = x ++ (i1_2 xs dir)

-- Process a run of implicit characters I1 - I2
implicit :: Run -> Run
implicit (LL xs) = LL (i1_2 (groupBy bidiEql xs) L)
implicit (LR xs) = LR (i1_2 (groupBy bidiEql xs) L)
implicit (RL xs) = RL (i1_2 (groupBy bidiEql xs) R)
implicit (RR xs) = RR (i1_2 (groupBy bidiEql xs) R)

-- If a run is odd (L) then reverse the characters
reverseRun :: [Level] -> [Level]
reverseRun [] = []
reverseRun (x:xs)
  | even (level x) = x:xs
  | otherwise = reverse (x:xs)

reverseLevels :: [[Level]] -> [[Level]] -> Int -> [[Level]]
reverseLevels w [] _ = w
reverseLevels w (x:xs) a = if (level (head x)) >= a
  then reverseLevels (x:w) xs a
  else w ++ [x] ++ (reverseLevels [] xs a)

-- Rule L2 Reorder
reorder:: [Run] -> Bidi -> [[Level]]
reorder xs base = foldl (reverseLevels []) runs levels
  where
  flat = concat (map toLevel xs)
  runs = map reverseRun (groupBy levelEql flat)
  levels = getLevels runs

-- Rule L4 Mirrors
mirror:: [Level] -> [Level]
mirror [] = []
mirror ((x,y,R):xs) = case getMirror y of
            Nothing -> (x,y,R):(mirror xs)
            Just a -> (x,a,R):(mirror xs)
mirror (x:xs) = x:(mirror xs)

logicalToDisplay :: [Attributed] -> [Ucs4]
logicalToDisplay attribs
  =let baseLevel = p2_3 attribs in
    let baseDir = (if odd baseLevel then R else L) in
    let x = explicit baseLevel attribs in
    let w = map weak x in
    let n = map neutral w in
    let i = map implicit n in
    map character (mirror (concat (reorder i baseDir)))
```