Multibyte character string processing in Scheme

Shiro Kawai
Scheme Arts, L.L.C.
shiro@schemearts.com

ABSTRACT

Nowadays, handling a large character set in application programs is a common requirement, and general-purpose programming languages are expected to support it naturally and efficiently.

The start line of supporting a large number of characters is to allocate more than one byte per character. However, it opens a can of worm. Actually, there are too many external factors that affect design decisions, and the discussion on this topic tends to diverge.

In this paper we break down the problem, and consider three issues. What are the advantages and disadvantages of two general implementation strategies, *wide character* and *multibyte* strings? To what extent should the language support for the character encoding issue? And what do we need to write a portable string manipulation routines that work well on both wide-character and multibyte implementations?

We are particularly interested in multibyte implementation, for it requires a slightly different way of programming from the traditional "string is an array of characters" model. We show that it is indeed possible to implement a multibyte string processing layer that works efficiently, and suggest several Scheme library APIs that allow a programmer to take advantage of these optimized implementations of multibyte string routines.

1. INTRODUCTION

A character string is traditionally considered as a simple array of characters, which was practically equivalent to a simple array of bytes until recently. However, it has become common for real-world applications to deal with a large character set, and the modern programming languages are expected to support it.

Some programming languages, including Java, specifies particular character set and encoding for internal processing. Some other languages don't specify particular encoding, but impose certain restrictions for the underlying character/string implementation, such that a character is represented by 16-bits, for example[2]. In Common Lisp, a string is a subtype of an array, so the n-th character in a string is expected to be accessed in $O(1)$ time, while it does not restrict the actual representation of a character object.

The Scheme standard[4] doesn't say anything about the underlying implementation of a string except it is a sequence of characters. This allows an implementor to choose whichever internal representation that satisfies the implementation's target criteria. Since the optimal internal representation may differ among different applications, this freedom of choice is an advantage.

There is a pitfall, though. It is tempting to write a code that does "optimization" based on an assumption of underlying representation of a string. If the code is intended to be portable, such optimization should be carefully avoided.

We are developing a Scheme implementation called Gauche, which is aiming at "scripting in daily life", for example, to process the chores like writing a throw-away program to scan log files easily. It is vitally important for us to handle a large character set natively, without burdening programmers by concerns like their characters being chopped up in the middle of string operations. We have chosen a *multibyte string* as the internal representation, in which each character in a string may use variable number of octets. Using Gauche, we noticed some patterns of string manipulation that work well for both multibyte strings and simple character arrays, while some patterns not.

In this paper, we discuss the implementation strategies of large character sets in the programming languages, particularly focusing on the comparison of multibyte strings and simple character arrays, and show a pattern that works independent from the underlying implementation.

In the next section we briefly explains the general architecture of multilingual applications, and discuss the extent that programming language should support them. In the following section, we discuss advantages and disadvantages of using wide-character and multibyte string as internal representation, and the issues in character encoding conversion. Then we explain the actual implementation strategy of Gauche. In the final section, we discuss a coding convention

To be appeared in the Proceedings of Second International Lisp Conference, October 2003, New York, NY.

of string handling which works well for both wide-character strings and multibyte strings, and conclude the paper.

2. ARCHITECTURE OF MULTILINGUAL APPLICATIONS

2.1 Handling large character sets

Adding support of a large character set to the language implementation is more than just allocating more bytes for each character. There can be more than one ways to represent a character. The rule to represent each character in a certain character set is called *Character Encoding Scheme*, or *CES*, and it is very common for an application program to deal with more than one CES.

Some CES's use variable number of bytes to represent each character (*multibyte* format), while other CES's use fixed width characters (*wide character* format).

It is usually simpler if you stick to use single encoding (*internal encoding*) inside the application. and convert from/to the encoding required by outside world (*external encoding*) when you do I/O or external library calls.

Most of external data exist in the multibyte character format. It is usually more compact than wide characters, and compatible to the huge amount of legacy data. Furthermore, multibyte character format doesn't depend on the byte-order of the architecture.

On the other hand, it is debatable whether internal encoding should use multibyte or wide-character format. Both has its own advantages and disadvantages. We'll discuss it in details in Section 3.

2.2 Layers of large-character set support

The difficult part of supporting large character set in a programming language is to decide where to draw a line. Large character set support can be arbitrary complex matter, if you start thinking language-sensitive character manipulation (in some languages it is difficult even to decide what a character is). On the other hand, you can support minimal features and shift the burden of complicated stuff to application writers. Such implementations are, however, less attractive nowadays.

For our discussion, it is useful to model the application in layers, as shown in figure 1.

The bottom layer, binary representation, directly touches the underlying bitpattern of the characters. Data I/O routines have to deal with this layer. This layer is also responsible for converting whatever external encodings to the internal encoding and vice versa.

The second layer deals with characters and strings as the data types defined in the programming language. This layer hides the bitpattern of characters, and provides an abstract "character" object to the above layers.

Usually the second layer is independent from the context of natural languages. A character won't change its semantics in the programming language by whether it is a part

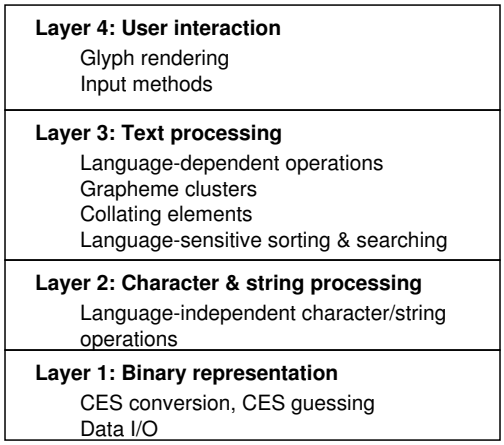

Figure 1: layers of multilingual system.

of Japanese text or Chinese text. Note that a character in this layer may not have one-to-one mapping to what a human user thinks as a character. So, arbitrary combination of characters in this layer may make a character string that doesn't mean anything, or even isn't renderable, as a text in a given natural language. Nevertheless, they are legal as a string in this layer.

On top of the characters and strings the programming language provides, the third layer, text processing, can be constructed, which is aware of the natural language context. This layer deals with grapheme clusters[11] and collating elements[10]. Language-sensitive sorting and searching has to be done in this layer as well.

If an application interacts with a human user, it adds another very complicated layer. Both rendering text using glyphs and handling input from the user largely depend on the natural language and context surrounding the text. However, it is beyond the scope of this paper.

The distinction between character/strings layer (layer 2) and text layer (layer 3) is important, since implementing text manipulation routines for all natural languages are extremely difficult task, if not impossible, thus the support of text layer may vary greatly among different implementations. The Unicode consortium, for example, discusses text processing in great detail[9], but it allows implementors a choice not to support particular characters and text processing rules related to it[1]

This means even if the language standard library provides the functionalities of the text layer, it is likely that the application will not be portable among implementations.

There is another important component which tends to be

[1]However, if you state you support particular Unicode characters, you have to implement rules described in the Unicode standard. One convenient way that is often taken is not to mention supporting particular Unicode characters, but just say a character can contain certain number of bits (which happen to be a Unicode in certain cases).

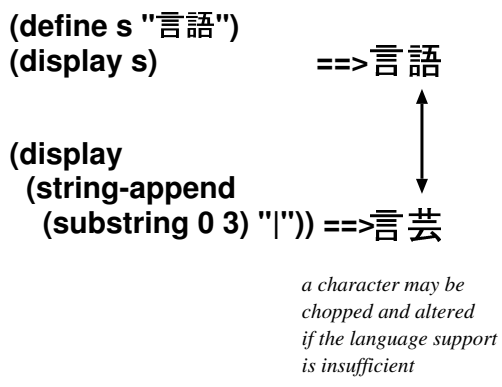

Figure 2: chopped and altered character.

missed in the discussion of large character set support, nevertheless is indispensable in practical applications: code guessing. It is very common that your application doesn't have prior knowledge of the encoding of external data, so you need to find out its encoding. Since many external encodings use the same binary representation for different purposes, this is an ill-defined problem. To get a good result, you need some context about the natural languages you are dealing with, while the guessing routine itself deals with binary representation of the data.

Now, the important question for the programming language implementors is how to realize the second layer abstraction, so that it will be easy to write the top layers, as well as to interface efficiently with the bottom layer.

2.3 Required programming language support

Some languages, like C, provide very little in the second layer and expose underlying representations. From the programmer's point of view, however, it is highly desirable that the language provides the following properties.

2.3.1 A character as an atomic unit

The character abstraction should be an atomic unit, in the sense that arbitrary combination of those characters will never produce an "illegal" string that causes a trouble in the system level.

That is, `string-ref` always returns a valid character in the string (if the index is within the range), `string-set!` never produces a corrupted string, and `string-contains` always returns a valid result. It sounds simple, yet there are lots of implementations that doesn't work in this way, annoying application programmers who have to deal with a large character set. For example, you can no longer casually trim a long diagnostic message to fit in a screen (see figure 2), or can't search strings to find a match.

To make matter worse, if one routine from a library depends on those multibyte-unaware functions, the entire library may become unusable for practical applications. For that reason, providing different set of "multibyte string API" from the language's native string API, should be avoided.

2.3.2 Back door for binary representation

On the other hand, if a language totally abstracts the underlying bitpattern of characters away from a programmer, it becomes impossible for him/her to write an important component of application such as CES conversion and guessing routine.

Although the language implementor can provide those features as built-in libraries, there are always the case that the provided one isn't enough and an application writer has to tweak it, or write his/her own[2].

3. MULTIBYTE VS WIDE CHARACTERS

One of the biggest design choice for language implementors is to use whether wide character or multibyte format in the string representation. It matters because it affects all over the place of the code, and it's not easy to switch back and forth. It has been rich source of debate in the mailing list and discussion groups of various languages, and each language made different choices: Python seems to decide for wide-character, while Tcl, Perl and Ruby took multibyte.

It is not easy to compare two approaches quantitatively, for it depends on lots of unknown factors. In this section, we try to summarize pros and cons of both approaches.

3.1 Wide characters

At the first glance, the obvious advantage of O(1) access time of wide characters is indispensable. However, It is so only if you can use "true" wide character representation, in which *all* characters are exactly the same length, and 1-to-1 mapping for external representation exists. In practice, it is not easy.

Unicode seems a good candidate of wide-character representation, and is indeed adopted by languages like Java. However, 16bit-a-character format, UTF-16, is no longer fixed-length since the adoption of surrogate pairs[3].

So, if you choose Unicode for wide-character internal representation, UCS-4 is the only choice. It seems to waste a lot of memory, since Unicode will only use up to 21-bits (U+10FFFF). However, extrapolating the current trend of increasing memory size and bandwidth, such waste can be tolerated in favor of $O(1)$ efficiency except special applications such as portable devices. Besides, it leaves you a room to extend your character type using extra bits.

There is a bigger problem than the size, however. Currently large part of external world doesn't take wide-characters; legacy libraries, operating system services, and stream I/Os. It can take very long time for them to support true UCS-4, since they have to provide double APIs unless everyone jumps to UCS-4 world at once. It is more likely that libraries start extending their API to use multibyte format such as

[2]One of the reason to do so is the incompatibility of conversion table. Also the code guessing algorithm largely depends on the data set the application deals with.

[3]The fact that it is rather "special" cases to see those characters doesn't justify applications to ignore those cases; if you do so, your application will suffer for a hard-to-track problem which appears in rare cases.

UTF-8[4]. And external encodings will likely remain to be multibyte format, after all.

It means you may end up keep converting internal wide-character and external multibyte format back and forth. Not only the overhead may be a problem, but also such conversion tends to cause so-called *round-trip problem*, that a character's identity isn't preserved if you convert it to another CES then convert back to the original CES.

Although the round-trip problem occurs regardless of whether internal encoding is wide-character or multibyte, it is less likely if you can avoid conversion at all, and multibyte internal encoding is easier to do so.

A good news for wide-character approach is that we'll likely to have extra CPU time and memory bandwidth in near future, and it can evolve into *fat character* approach—you don't use fixed bitpattern for characters, but treat each character as an individual object that may contain arbitrary information. Round-trip problem can be solved if each character *knows* where it came from. There are already some attempt to move to that direction, such as CHISE project[5]

It should also be noted that wide character or fat character string doesn't need to have $O(1)$ access-time property. In some languages, such as Haskell or Arc, a string is just a list of characters, hence has $O(n)$ access-time. Except that, however, it does share the properties of wide-character strings. For example, you can still use fast string search algorithms such as Boyer-Moore, and the mutating strings (if the language ever allows) doesn't cause so much problems as in multibyte strings as discussed below.

3.2 Multibyte string

Using multibyte format for internal encoding is to give up $O(1)$ string access for better interoperability with external world. If you know you deal with particular external multibyte encoding very often, which is common, then having the same multibyte encoding as internal encoding can eliminate hairy character-conversions. It is also memory-efficient, which matters for small devices with limited resources.

Whether multibyte format is better choice than wide-character format or not largely depends on how much you think the following properties as disadvantages:

- $O(n)$ index-access
- Cost of character boundary identification
- High-cost of mutating strings

The penalty from these disadvantages can be avoided to certain extent, if the implementor provides a certain set of higher-level string procedures, and a programmer carefully avoids the operations that assume the "character array" model of the string.

3.2.1 $O(n)$ access

If you've been using indexed string access all the time, $O(n)$ access may sound unacceptable burden. However, it largely depends on what type of application you're writing.

Unless you are writing some complicated text analysis algorithm, typical string access patterns are either:

1. to retrieve sequentially one character at a time, or
2. to extract a part of the string using some searching library functions, like regular expression matcher or `string-contains` in SRFI-13[7].

Sequential access of the string doesn't need an indexed access; all we need is a sort of string iterator, which can keep the pointer to the current character boundary in the multibyte string. It doesn't cost $O(n)$ to retrieve the next character, and it has advantage over indexed access since boundary-check is easier.

Some efficient string searching algorithms need operations such as "skip n characters", which can be slower if you proceed character by character. However, with some CES you may be able to apply those algorithms in *byte sequence* of the strings. If the internal encoding is UTF-8, for example, you can directly use Boyer-Moore algorithm on binary representation of the string. Regular-expression matching engine can also be optimized to operate byte stream whenever possible. The language implementor can provide a set of string searching primitives that take advantage of the underlying CES, so that users of the library don't always need to pay for $O(n)$ cost.

$O(n)$ may become a big problem if you are dealing with very large strings. In the structured-document world (read 'XML'), it is less likely that you see one megabyte of a single chunk of string. If you do need to handle such strings, however, that may be a sign that what you really need is a data structure like a vector. For example, you may have a long DNA sequence represented by a string and want to hop around in it by indexes; then it may be suitable to use `u8vector` of SRFI-4 instead, for it does provide guaranteed $O(1)$ access and packed byte array, unlike a sort of opaque structure of strings.

3.2.2 Character boundary

This effect isn't visible for the programmers that uses the language, but matters for implementors.

UTF-8 has a nice property that you can tell whether a byte in the string is a beginning of a multibyte character or not without looking at other bytes in the string. Some other multibyte CES, such as Shift-JIS or EUC (packed format), doesn't have such a property. They have certain bytes which can be both the first byte and the following byte of a multibyte character.

[4] Actually, some functions may not care whether the passed string is multibyte or just a single-byte sequence. It makes library authors less eager to provide duplicated functionality in wide-character format

[5] CHISE project[8][6] http://www.kanji.zinbun.kyoto-u.ac.jp/projects/chise/index.html

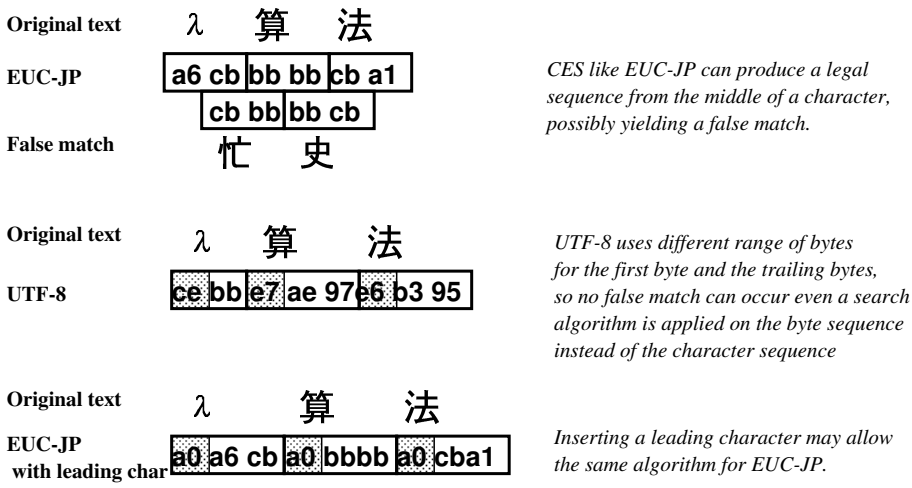

Figure 3: False match.

This prevents the implementors from using efficient string-searching algorithms directly on the underlying byte sequence, since it may yield a false match (figure **??**). An awkward workaround is to insert a "leading character" before every character. The leading character must differ for any byte that can be a part of valid multibyte character[6].

3.2.3 Mutating strings

What happens if you substitute a character in a multibyte string for another character, by `string-set!`, but the two characters have different number of bytes? You have to reallocate the entire string, while keeping the identity of the string (in `eq?` sense) before and after mutation.

Generally this leads you to allocate the body of the string (byte array) separately from the string object itself. When a string is mutated in the way that it changes the size of the string, the body is reallocated, the original contents are copied except the mutated character is substituted, then the string object is updated to point to the new body.

This sounds terribly inefficient. However, how often you need to use `string-set!`? Its most common use is in construction of a string; you first allocate a certain length of string by `make-string`, then fill it with one character at a time. Even when the length of constructing string is unknown, you can use the allocated string as if it is a buffer, and chain them together until you finish, then append all the buffers into a result string.

This kind "optimization" is based on an assumption that string is a simple array of characters, which is, in fact, implementation-dependent.

To construct a string sequentially, we already have SRFI-6's output string port[1]. SRFI-13 also provides some functional string constructors. The implementor can support these procedures natively, taking advantage of underlying representation.

There is a case that you want to mutate a single character of a existing string, but it is a special case of generic string replacing where the original part and the replaced part happens to be a single character. It isn't very common in applications.

If we exclude above cases, the rest of `string-set!` is really the cases that use a string as some sort of data structure where each character have some meaning. Again, vectors or records might be more suitable for such purpose.

4. CES CONVERSION PROBLEMS

It is better that you can avoid converting encodings whenever possible. Not only because it has overhead, but also because mapping between two CES's is often ill-defined, resulting a kind of problems that don't appear in labs but bite you hard in production environments.

In this section we examine common problems related to code conversion that implementors have to keep eyes on.

4.1 Where conversion happens

If you have the single internal encoding, the natural place to implement CES conversion is a port; the other side of a port is usually an external world, where people speak in external encoding. However, when you read from the port, you always get a character in the internal encoding.

To have a CES converter in a port has a big impact on the implementation of `char-ready?`.

To be strict, `char-ready?` should return `#t` only if the port has a whole character, so that the subsequent `read-char` will never block. However, if the port does a CES conversion, it can be arbitrary expensive for `char-ready?` to check if the available data can be converted to a whole character in internal encoding.

Suppose that the external encoding is a stateful encoding

[6] One possible optimization is to leave characters in ASCII range as is, and use a leading character for other characters.

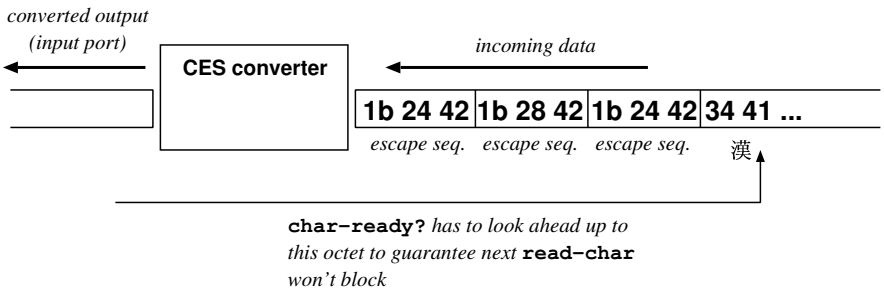

Figure 4: `char-ready?` with CES conversion port.

such as ISO-2022[3] and you are reading from the port. Input may contain arbitrary number of escape sequences (it is unusual, nevertheless it is legal). In order for `char-ready?` to find out whether the converter output produces a character, it needs to lookahead arbitrary amount of input data. When `char-ready?` finally finds out the a character is ready, the data has actually been read4.

Alternatively, `char-ready?` can just check whatever data is available at the port endpoint, but not read the data at all, much like what Unix's select(2) or poll(2) system call does. It should work mostly well, for in many cases a chunk of bytes that represent a character arrive to the port at once. However, it doesn't eliminate the possibility of the subsequent `read-char` to hang, hence violates R5RS.

There may be another strategy: restrict ports to read from the internal encoding when it is used as character input. For other encodings, a port works only with special binary I/O primitives (like `read-byte` and `write-byte` in Common Lisp, or something like `read-block` and `write-block` to read/write a chunk of binary data), and a separate conversion procedure does conversion between binary sequences and character sequences. If the implementation also provide a "procedural port", where the user can implement his/her own port by providing handler procedures, then he/she can implement a port with conversion on top of such restricted ports.

It is also required that you can switch the external encoding of the conversion port in some way. It doesn't need to support switching between arbitrary encodings—which may cause a complicated problem if one of the encodings is stateful—but at least the implementation have to switch some default CES (which can be ASCII) to other CES. Suppose you are reading an XML document. Before you start any conversion, you have to read `<?xml version="1.0" encoding="...">` to know what encoding the document is written in.

4.2 Undefined or illegal sequence

The external world is not a perfect world, and it is *very often* that you receive a byte sequence which can't happen in legal sequences of supposed external encoding. How to handle such a sequence may differ depending on what application wants. There can be three scenarios:

- Signal an error and interrupt the processing.
- Substitute the invalid sequence to other valid character sequence.
- Silently ignore the invalid sequence and restart conversion from where the valid sequence begins.

It is desirable that the implementation provides some way to specify a handler for the conversion mechanism so that the application programmer can choose a suitable strategy.

Note that such handler may want to examine the context where the illegal input sequence occurs. For example, the handler looks ahead the input stream, and if it sees the corruption is local, it lets conversion process to continue by ignoring the illegal sequence, but if it finds out the following sequence is totally garbage, it raises an error.

If a port does conversion, however, implementing such a look ahead mechanism adds large complexity to the port internals.

4.3 Round-trip problem

Round-trip problem is that when you convert a character from CES A to CES B, then convert back to CES A, you get a different character. This appends because conversion between CES A and CES B are not 1-to-1 mapping.

To make matter worse, the mapping is often ill-defined, so the same program may exhibit different behavior among implementations. Sometimes the mapping tables are just different. Sometimes one implementation doesn't define mapping of the character between CES A and CES B, and the application uses the handler mechanism of undefined sequence described above to introduce a substitution character. If you port that program to another implementation that does define the mapping, you suddenly get a different behavior.

This is not a rare problem. Indeed, it *always* happens when you're writing applications in multilingual environment.

One way to avoid the risk of this problem is to avoid conversion from the first place—use the same internal encoding as the supposed external encoding. It is not practical to support internal encodings for every possible external encoding,

but it is often the case that you know the primary encodings used in the production environment. The implementation may provide a run-time flag that switches the internal encoding, or provide different binaries compiled with different switches.

When implementing a conversion routine, it is tempting to choose one CES as a "pivot" CES, and convert any incoming CES to the pivot, then convert it to the outgoing CES. However, it isn't necessarily work for all combinations. Sometimes you need to use different pivot CES.

4.4 Code guessing

Another common problem with multiple CES's is that you always receive a data whose encoding is unknown, so you have to guess its CES.

It is an ill-defined problem. Many CES's use code ranges that overlap each other, so a byte sequence can often be totally valid character sequences of multiple CES's.

So the perfect guess can't be achieved. You have to use whatever clues available at your hand. If you know the natural language the content is written in, it greatly reduces the candidate CES's. In practice, it is often the case.

A naive algorithm looks for a characteristic byte sequence that happens only in one of the candidate CES's. For example, if you know the data is either in one of ISO-2022-JP, Shift_JIS or EUC-JP (in JISX0201 and JISX0208), you can use something like the following logic[5] (the logic is simplified here for concise explanation).

- If you see escape sequences like $1B_{16}$ '$' 'B', it is ISO-2022-JP.
- If you see code between 81_{16} to $8D_{16}$ or 90_{16} to $9F_{16}$, or $A1_{16}$ to DF_{16} followed by 40_{16} to $9F_{16}$, or $E0_{16}$ to EF_{16} followed by 40_{16} to $A0_{16}$, it is Shift_JIS.
- If you see code between $F0_{16}$ to FE_{16}, or $A1_{16}$ to DF_{16} followed by $F0_{16}$ to FE_{16}, or $E0_{16}$ to EF_{16} followed by FD_{16} or FE_{16}, it is EUC-JP.

Unfortunately, if you have a newer CES that tends to spread to wider code area, and/or you have more candidate CES's, this naive deterministic approach easily falls to unusable. For example, simply using the newer standard JISX0213:2000, which adds a lot more code points, and adding UTF-8 to the candidate CES's, the above approach can't determine the code for large part of valid Japanese text.

Better approach is to use some kind of probabilistic state machine. Sensible text in the natural language is a small part of possible combination of all the characters, so the statistical approach is likely to work in most cases. It is still also difficult, though, if the input data is very small (it often happens when processing the web form input in cgi scripts).

From the language implementor's view, note the two facts: First, the guessing routine requires set of candidate CES's. That means it can't be totally hidden "under the hood" of the binary representation layer in figure 1, but requires some input from the text processing layer of the application. Secondly, in order to implement this in the language, you need a mechanism to manipulate binary chunk of data, and attach it to the port as a buffer if you use a port to do real conversion afterwords.

5. IMPLEMENTATION EXAMPLE: GAUCHE

In this section, we describe how Gauche implements multibyte string natively, trying to keep efficiency as much as possible, and also what we learned from it.

5.1 String objects

Gauche's string object consists of an anchor object and the backing storage. The body of multibyte string is stored in the backing storage, and the anchor object has the following slots:

- *length*, the number of characters in it
- *size*, the number of bytes it occupies
- pointer to the backing storage

Having both length and size have several pleasant properties. For example, it is easy to test whether the string contains only single-byte characters or not. If *length* = *size*, we can use $O(1)$ access. So, if most of the strings the application deals with are in ASCII, there's little overhead. Operations such as string concatenation and string index boundary check can be done easily as well. For Scheme level, Gauche has `string-size` procedure that returns the size of the string in bytes.

Strings are managed by copy-on-write policy; whenever a string is mutated, a fresh backing storage is allocated and the original string is copied, except the altered part is filled by the new character.

This allows us to share a backing storage by more than one strings; for example, `substring` allocates only the anchor object and points into the original string.

Note that it eliminates the need of indexed access when you have a structure like suffix array. Instead of having a string and array of indexes, you can have just an array of strings that actually share the back-end storage.

The internal encoding can be changed at compile-time.

So far this strategy seems working reasonably. The performance bottleneck doesn't usually come from multibyte string, but from elsewhere, like port locking overhead or a poorly implemented stack-overflow handler. There are cases that the third-party library runs slow, and we find it uses `string-set!` extensively to do the "buffering". Usually rewriting it by using SRFI-6 string ports yields cleaner code and better performance.

5.2 Character objects

Gauche's character object is just like a small integer packed in a word with the tag. Gauche reserves 29bits to represent a character on 32bits/word architecture.

In the current design, a character is converted to a "packed" format when read from multibyte string. That is, if a character in a string is represented by a UTF-8 multibyte sequence $\texttt{E7}_{16}$ $\texttt{AE}_{16}$ 97_{16}, `string-ref` converts it to a UCS-4 format $\texttt{7B97}_{16}$. When it is stored in a string, the character is unpacked to a UTF-8 sequence.

It turned out that such a conversion have non negligible overhead, especially the packing/unpacking operation is rather complex in Unicode. (EUC-JP or Shift_JIS has much more simple operation, which is just shift and add).

Since the bitpattern in a character object doesn't matter to outside world, we're planning to switch to a simple shift and add packing even for Unicode. Maximum code position in Unicode in utf-8 is $\texttt{F4}_{16}$ $\texttt{8F}_{16}$ $\texttt{BF}_{16}$ $\texttt{BF}_{16}$, but we can just drop top 3 bits by left shift, and recover them by arithmetic right shift (because in 4 octet format, the first byte range is between $\texttt{F0}_{16}$ and $\texttt{F4}_{16}$).

5.3 Scanning and matching

Gauche provides a built-in string search function (`string-scan` and `string-split`) and regular-expression matching engine, which implement optimization strategies described in "Multibyte string" section above. Built-in regular expression compiler takes multibyte POSIX-style regular expression description and produces an NFA that works on byte stream[7]. Therefore, there is little penalty of using multibyte internal encodings.

An important feature of those native searching routines are that they can return matched substring directly instead of the index to point the match. Since the underlying routine knows exact location of match in the string, returning substring bypasses $O(n)$ indexing.

For example, the regular-expression matcher returns an opaque match object, instead of explicit list of indexes. The match object actually keeps the pointer to the backing storage, so extracting matched substring is much faster than extracting indexes and calling `substring`. In practice, you hardly use matched indexes other than passing them to `substring`, so such a shortcut is indeed an advantage.

5.4 String iterator

We found that a string port abstraction as in SRFI-6 is sometimes not enough to implement procedures of SRFI-13 efficiently. The features SRFI-6 string port lacks are the ability to retrieve efficiently the rest of the content of input string port, and the ability to iterate a string in reverse order.

We introduced a low-level string iterator object called string pointers, which keeps the string object and a direct pointer into the backing storage.

[7]The regexp engine cares about "character" only when it sees a character range expression that contains multibyte characters.

However, we have a feeling that introducing such an ad-hoc object is awkward. Most of the function of string pointer can be abstracted by adding utility API for string ports.

5.5 Input and output

We implemented CES conversion in the port. Actually, the conversion port works as if a kind of filter—it takes a source port (when it is an input port) or a destination port (when it is an output port) at construction time. If it is an input port, the conversion port reads data from the original port, and returns a converted result to the caller. If it is an output port, the conversion port takes a data from the output procedures, then push out the converted result to the original output port.

The behavior of "switching the encoding" described in the CES conversion problems can be realized by wrapping the original port by appropriate conversion port after deciding what the external encoding is.

Gauche's port also supports binary I/O, and allows users to mix binary I/O and character I/O. To support it, a port has a small scratch buffer that is used to decompose or build multibyte character when required. For example, suppose you peek one character, then read one byte. The peeked character is stored in the scratch buffer, and next `read-byte` retrieves one byte from the buffer.

The mechanism of adding handlers for undefined or illegal byte sequences is not implemented yet, but there are enough requests to make us implement it pretty soon.

5.6 Byte strings

When you are working in low-level stuff, sometimes you need to construct a string from bit-level representations. Sometimes you might even deal with a string that is not a valid multibyte string in the internal encoding. In order to do so, we actually have two types of strings, one is normal character string (simply referred as a string) and the other is a byte string.

We introduced a byte string in very early stage, in order to represent a state when internal routines created an incomplete multibyte string.

However, now we are not certain if we really need it. In order to play with bits, we have SRFI-4 vectors. Gauche provides `string->u8vector` and alike, to extract binary representation of the string as a SRFI-4 vector, and pack it back to a string. And having two kind of strings is a source of confusion.

There are still an issue to solve: we have string ports, and we have binary I/O. What if we output a byte sequence to an output string port that doesn't create a legal multibyte string?

6. TOWARD PORTABLE STRING API

Scheme leaves lots of implementation details to the language implementors, and we think it is a good thing. You need a different strategy to solve different problems.

However, it will be useful to have a set of string APIs that abstract underlying representations enough so that implementations can provide optimized versions.

6.1 Bad habits

These are some Scheme idioms that make some assumptions in the underlying string implementation, therefore may not work well on some implementations.

6.1.1 Using string as a buffer

Allocating a string and filling it by `string-set!` is a bad idea. It behaves poorly in multibyte implementations, and even in wide-character implementations, every `string-set!` may involve boundary check and/or trigger a write barrier.

Output string port can be much easier to use, and gives lots of optimization opportunity for implementors. For functionally minded, SRFI-13's string constructor should give cleaner model.

The only possibility that you ever need `string-set!` to construct a string is somehow you need to fill it in non-sequential order. If the algorithm requires it somehow, you can go with a vector and converts it to a string.

6.1.2 Using string as a byte vector

Sometimes people think a vector take too much space (a word per each item) and use a string as a sort of byte vector (a byte per each item). It was also the way to pretend doing binary I/O in Scheme. It promptly breaks when the underlying implementation starts supporting a large character set. We have SRFI-4 now.

We also feel that stopping this usage eliminates another source of `string-set!`.

6.1.3 Indexed access

If you access a string sequentially, string ports or SRFI-13 mapping functions usually gives cleaner code than using indexed access.

If the algorithm absolutely requires random access in a string, you may want to consider using a vector instead.

6.2 Useful additional APIs

6.2.1 Ports

We need at least `read-byte` and `write-byte` to distinguish binary I/O from character I/O. For the sake of efficiency, some sort of block I/O will be also handy. We can use a SRFI-4 vector as the buffer to be used in such block I/O.

Strictly speaking, we also need `byte-ready?` as a binary version of `char-ready?`, since `char-ready?` might need to look ahead more than one bytes.

The semantics of mixing binary I/O and character I/O in a port can be complicated, but we can leave it to the implementors.

6.2.2 String ports

Two auxiliary APIs will be useful. One to take the content remaining in the input string port as a string, and the other creates an input string that traverses the given string in reverse order. Those APIs can easily be written in Scheme, but the implementors can provide optimized version.

6.2.3 Opaque string pointer

It is useful not to specify what is returned from string searching functions. Instead it can be an object p, which can be used to retrieve a substring of the original string s, when passed to a substring extraction function with s. For example, we could have a function *extract-after* which returns a substring after p in s, when it is called like (*extract-after* s p).

On implementations with $O(1)$ access-time string, p can be just an integer, and *extract-after* is simply `string-drop`. On multibyte implementations, p can be an object that has a pointer into the middle of s.

Alternatively, we could just extend the existing API of `substring` and SRFI-13 strings so that they accept this opaque string pointer in place of index.

6.2.4 Manipulating binary representation

There has to be a way for programmers to work on lower-level representation of a string. So that he/she can write CES conversion routine and CES guessing routine, for example.

It is important that the "view" of binary representation should differ between wide-character and multibyte string format. For wide-character string, the binary view would be an array of bit-vectors (which can be an integer), where each bit-vector is at least as wide as the character's internal representation is. On the other hand, multibyte string's view would be just an array of bytes. For portability, the former can be `u32vector` and the latter can be `u8vector`, i.e. we'll have `string->u32vector` and `u32vector->string`, for wide characters, and `string->u8vector`, and `u8vector->string` for multibyte strings. The implementations can provide both for compatibility; converting from one representation to another is easy.

Note that if the implementation uses "fat character" representation, whole idea of binary representation becomes irrelevant. So this APIs don't really aim at perfect portability, but gives some common ground to hack into internals of strings.

6.2.5 Code point and character

`integer->char` and `char->integer` don't specify concrete mapping between integers and characters. Obviously, something like `ucs->char` and `char->ucs`[8] can be introduced for handy conversion between a character and a portable integer code point. However, `ucs->char` and `char->ucs` doesn't always work.

- A single Unicode code point may be mapped to a sequence of characters. This is crucial to support XML's

[8] The prefix '`ucs`' is taken from SRFI-14's `ucs-range->char-set`.

character entity reference on the Scheme implementation that doesn't yet support full Unicode characters, for example.

The entity reference `&#3042;` may be expanded to a list equivalent to `(map integer->char '(#xe3 #x81 #x82))`, for the implementation that just have one-byte per character string to pretend it supports UTF-8.

The entity reference `&#10302` may be expanded to a list equivalent to `(map integer->char '(#xd800 #xdf02))`, for the implementation that only supports UTF-16 and requires a surrogate pair to represent the specified character.

- A single Scheme character may be mapped to a sequence of Unicode code points. For example, a character of code $\texttt{A4F7}_{16}$ in EUC-JP encoding of JISX0213:2000 doesn't have a corresponding character in Unicode, and it has to be represented by one Unicode character with one Unicode combining character, `U+304B` and `U+309A`.

We can have `ucs->char-list` and `char->ucs-list`, respectively, to absorb this difference.

We may also need the reverse functions of those, but the context where the reverse functions are called will depend on implementations. That is, an implementation that uses one-byte per character and doesn't support UTF-8 may need to call the reverse function with a list of three characters, whose internal code points are `#xe3` `#x81` and `#x82`, in order to produce a Unicode code point `#3042`, while an implementation that fully supports UCS-4 can just pass a single character `U+3042`. So, when calling the reverse functions, the programmer has to code explicitly in implementation-dependent way.

7. CONCLUSION

We have discussed various aspects of supporting a large character set in Scheme. First we modeled the architecture of large character set support, and considered the extent that the programming language needs to provide. Then we compare advantages and disadvantages of two major formats of internal encodings, namely wide-character and multibyte strings. The traditional naive "array of characters" model of string strongly suggests wide-character string is more straightforward, but we show there are various techniques that can make multibyte strings a feasible choice. We also discussed pitfalls of another big source of problems, CES conversion. We then showed how we realized multibyte string in our Scheme implementation, Gauche, and reflected on our design choices. Finally, we made some suggestions for Scheme programmers and implementors in order to write a portable programs that perform well independent from the internal encoding.

The issue of large character set support in a programming language is a part of internationalization (i18n) and multilingualization (m17n) efforts, which have been discussed for over a few decades. However, it seems that somehow the discussion on this issue tends to remain local within the community of a particular language, and tends to be scattered in mailing list archives and web articles.

By this paper, we aimed at consolidating such previous experiences, so that this paper can serve a reference and a starting point for those who are interested in this issue. The fact that Scheme has lots of implementations makes it ideal to try out different ideas.

As the future work, we are planning to do quantitative comparison between wide-character and multibyte strings, using application programs written in Gauche and being used in the production environment.

8. REFERENCES

[1] William D Clinger, SRFI-6: Basic String Ports, In *Scheme Request for Implementation*, `http://srfi.schemers.org/srfi-6/` .

[2] Richard Gillam, Adding internationalization support to the base standard for JavaScript: Lessons learned in internationalizing the ECMAScript standard, `IBM developerWorks, September 1999`, `http://www-106.ibm.com/developerworks/library/internationalization-support`

[3] *ISO/IEC 2022:1994 : Information technology – Character code structure and extension techniques.*

[4] R. Kelsey, W. Clinger, J. Rees (eds.), Revised^5 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices*, **33**(9), September, 1998.

[5] Ken Lunde, CJKV Information Processing, *O'Reilly & Associates, Inc*, 1999.

[6] MORO Shigeki, Software Review: CHISE Project, in *Journal of Japan Association for East Asian Text Processing (JAET)* No. 3, October 2002 (in Japanese).

[7] Olin Shivers, SRFI-13: String Libraries, In *Scheme Request for Implementation*, `http://srfi.schemers.org/srfi-13/` .

[8] MORIOKA Tomohiko, CHISE project—beyond the UTF-2000, *m17n2001: the Fifth International Symposium on Multilingual Information Processing and Open Source Software*, 2001.

[9] Unicode Consortium, Unicode Standard, Version 4.0.0, defined by: *The Unicode Standard, Version 4.0*, Reading, MA, Addison-Wesley, 2003. ISBN 0-321-18578-1, `http://www.unicode.org/versions/Unicode4.0.0/`

[10] Unicode Consortium, Unicode Technical Standard #10: Unicode Collation Algorithm, `http://www.unicode.org/unicode/reports/tr10/`, 2002.

[11] Unicode Consortium, Unicode Standard Annex #29: Text Boundaries, `http://www.unicode.org/unicode/reports/tr29/` .