Internationalization Features of R 2.1.0

by Brian D. Ripley

R 2.1.0 introduces a range of features to make it possible or easier to use R in languages other than English or American English: this process is known as *internationalization*, often abbreviated to *i18n* (since 18 letters are omitted). The process of adapting to a particular language, currency and so on is known as *localization* (abbreviated to *l10n*), and R 2.1.0 ships with several localizations and the ability to add others.

What is involved in supporting non-English languages? The basic elements are

1. The ability to represent words in the language. This needs support for the *encoding* used for the language (which might be OS-specific) as well as support for displaying the characters, both at the console and on graphical devices.

2. Manipulation of text in the language, by for example `grep()`. This may sound straightforward, but earlier versions of R worked with bytes and not characters: at least two bytes are needed to represent Chinese characters.

3. Translation of key components from English to the user's own language. Currently R supports the translation of diagnostic messages and the menus and dialogs of the Windows and MacOS X GUIs.

There are other aspects to internationalization, for example the support of international standard paper sizes rather than just those used in North America, different ways to represent dates and times,[1] monetary amounts and the representation of numbers.[2]

R has 'for ever' had some support for Western European languages, since several early members of the core team had native languages other than English. We have been discussing more comprehensive internationalization for a couple of years, and a group of users in Japan have been producing a modified version of R with support for Japanese. During those couple of years OS support for internationalization has become much more widespread and reliable, and the increasing prevalence of UTF-8[3] locales as standard in Linux distributions made greater internationalization support more pressing.

How successfully your R will support non-English languages depends on the level of operating system support, although as always the core team tries hard to make facilities available across all platforms. Windows NT took internationalization seriously from the mid 1990s, but took a different route (16-bit characters) from most later internationalization efforts. (We still have R users running the obsolete Windows 95/98/ME OSes, which had language-specific versions.) MacOS X is a hybrid of components which approach internationalization in different ways but the R maintainers for that platform have managed to smooth over many of its quirks. Recent Linux and other systems using `glibc` have extensive support. Commercial Unixes have varying amounts of support, although Solaris was a pioneer and often the model for Open Source implementations.

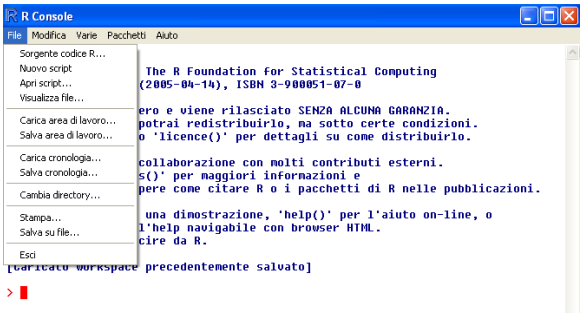

Figure 1: Part of the Windows console in an Italian locale.

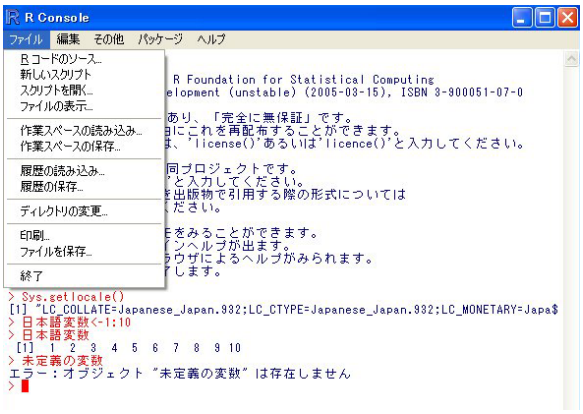

Figure 2: Part of the Windows console in Japanese. Note the use of Japanese variable names, that the last line is an error message in Japanese, as well as the difference in width between English and Japanese characters.

Hopefully, if your OS has enough support to allow you to work comfortably in your preferred language, it will have enough support to allow you to use R in that language. Figures 1 and 2 show examples of internationalization for the Windows GUI: note how the menus are translated, variable names are accepted in the preferred language, and error

[1] e.g. the use of 12-hour or 24-hour clock.
[2] for example, the use of , for the decimal point, and the use of , or . or nothing as the thousands separator.
[3] see below.

messages are translated too. The Japanese screenshot shows another difference: the Japanese characters are displayed at twice the width of the English ones, and cursor movement has to take this into account.

The rest of this article expands on the concepts and then some of the issues they raise for R users and R programmers. Quite a lot of background is needed to appreciate what users of very different languages need from the internationalization of R.

Locales

A *locale* is a specification of the user-specific environment within which an operating system is running an application program such as R. What exactly this covers is OS-specific, but key components are

- The set of 'legal' characters the user might input: this includes which characters are alphabetic and which are numeric. This is the C locale category `LC_CTYPE`.
- How characters are sorted: the C locale category `LC_COLLATE`. Even where languages share a character set they may sort words differently: it comes as a surprise to English-speaking visitors to Denmark to find 'Aa' sorted at the end of the alphabet.
- How to represent dates and times (C locale category `LC_TIME`). The meaning of times also depends on the timezone which is sometimes regarded as part of the locale (and sometimes not).
- Monetary formatting (C locale category `LC_MONETARY`).
- Number formatting (C locale category `LC_NUMERIC`).
- The preferred language in which to communicate with the user (for some systems, C locale category `LC_MESSAGES`).
- How characters in that language are encoded.

How these are specified is (once again) OS-specific. The first four categories can be set by `Sys.setlocale()`: the initial settings are taken from the OS and can be queried by calling `Sys.getlocale()` (see Figure 2). On Unix-like OSes the settings are taken from environment variables with the names given, defaulting to the value of `LC_ALL` and then of `LANG`. If none of these are set, the value is likely to be `C` which is usually implemented as the settings appropriate in the USA. Other aspects of the locale are reported by `Sys.localeconv()`.

Other OSes have other ways of specifying the locale: for example both Windows and MacOS X have listboxes in their user settings. This is becoming more common: when I select a 'Language' at the login screen for a session in Fedora Core 3 Linux I am actually selecting a locale, not just my preferred language.

R does not fully support `LC_NUMERIC`, and is unlikely ever to. Because the comma is used as the separator in argument lists, it would be impossible to parse expressions if it were also allowed as the decimal point. We do allow the decimal point to be specified in `scan()`, `read.table()` and `write.table()`. `Sys.setlocale()` does allow `LC_NUMERIC` to be set, with a warning and with inconsistent behaviour.

For the R user

The most important piece of advice is to specify your locale when asking questions, since R will behave differently in different locales. We have already seen experienced R users insisting on R-help that R has been broken when it seems they had changed locales. To be sure, quote the output of `Sys.getlocale()` and `localeToCharset()`.

For the package writer

Try not to write language-specific code. A package which was submitted to CRAN with variables named `años` worked for the author and the CRAN testers, but not on my Solaris systems. Use only ASCII characters[4] for your variable names, and as far as possible for your documentation.

Languages and character sets

What precisely do we mean by 'the preferred language'? Once again, there are many aspects to consider.

- The language, such as 'Chinese' or 'Spanish'. There is an international standard ISO 639[5] for two-letter abbreviations for languages, as well as some three-letter ones. These are pretty standard, except that for Hebrew which has been changed.
- Is this Spanish as spoken in Spain or in Latin America?
- ISO 639 specifies `no` as 'Norwegian', but Norway has two official languages, Bokmål and Nynorsk. Which is this?[6]

[4] Digits and upper- and lower-case A–Z, without accents.

[5] http://www.w3.org/WAI/ER/IG/ert/iso639.htm or http://www.loc.gov/standards/iso639-2/englangn.html

[6] Bokmål, usually, with `nn` for Nynorsk and `nb` for Bokmål specifically.

- The same language can be written in different ways. The most prominent example is Chinese, which has Simplified and Traditional orthography, and most readers can understand only one of them.
- Spelling. The Unix `spell` program has the following comment in the BUGS section of its `man` page:

  > 'British spelling was done by an American.'

  but that of itself is an example of cultural imperialism. The nations living on the British Isles do not consider themselves to speak 'British English' and do not agree on spelling: for example in general Chambers' dictionary (published in Scotland) prefers '-ise' and the Oxford English Dictionary prefers '-ize'.

To attempt to make the description more precise, the two-letter language code is often supplemented by a two-letter 'country or territory' code from ISO 3166[7]. So, for example

pt_BR is Portuguese as written in Brazil.

zh_CN is (simplified) Chinese as written in most of mainland China, and **zh_TW** is (traditional) Chinese as written in Taiwan (which is written in the same way as **zh_HK** used in Hong Kong).

en_US is American.

en_GB is English as written somewhere (unspecified) in the UK.

We need to specify the language for at least three purposes:

1. To delimit the set of characters which can be used, and which of those are 'legal' in object names.
2. To know the sorting order.
3. To decide what language to respond in.

In addition, we might need to know the direction of the language, but currently R only supports left-to-right processing of character strings.

The first two are part of the locale specification. Specifying the language to be used for messages is a little more complicated: a Nynorsk speaker might prefer Nynorsk then Bokmål then generic Norwegian then English, which can be expressed by setting `LANGUAGE=nn:nb:no` (since generic 'English' is the default).

Encodings

Computers work with bits and bytes, and *encoding* is the act of representing characters in the language as bytes, and *vice versa*. The earliest encodings (e.g. for Telex) just represented capital letters and numbers but this expanded to ASCII, which has 92 printable characters (upper- and lower-case letters, digits and punctuation) plus control codes, represented as bytes with the topmost bit as `0`. This is also the ISO 646 international standard and those characters are included in most[8] other encodings.

Bytes allow 256 rather than 128 different characters, and ISO 646 has been extended into the bytes with topmost bit as `1`, in many different ways. There is a series of standards ISO 8859-1 to ISO 8859-15 for such extensions, as well as many vendor-specific ones (such as the 'WinAnsi' and other code pages). Most languages have less than the 186 or more characters[9] that an 8-bit encoding provides. The problem is that given a sequence of bytes there is no way to know that it is in an 8-bit encoding let alone which one.

The CJK[10] languages have tens of thousands of characters. There have been many ways to represent such character sets, for example using shift sequences (like the shift, control, alt and altgr keys on your keyboard) to shift between 'pages' of characters. Windows uses a two-byte encoding for the ideographs, with the 'lead byte' with topmost bit `1` followed by a 'trail byte'. So the character sets used for CJK languages on Windows occupy one or two bytes.

A problem with such schemes is their lack of extensibility. What should we do when a new character comes along, notably the Euro? Most commonly, replace something which is thought to be uncommon (the generic currency symbol in ISO 8859-1 was replaced by the Euro in ISO 8859-15, amongst other changes).

Unicode

Being able to represent one language may not be enough: if for example we want to represent personal names we would like to be able to do so equally for American, Chinese, Greek, Arabic and Maori[11] speakers. So the idea emerged of a universal encoding, to represent all the characters we might like to

[7] http://www.iso.org/iso/en/prods-services/iso3166ma/index.html.

[8] In a common Japanese encoding, backslash is replaced by Yen. As this is the encoding used for Windows Japanese fonts, file paths look rather peculiar in a Japanese version of Windows.

[9] the rest are normally allocated to 'space' and control bytes

[10] Chinese, Japanese, Korean: known to Windows as 'East Asian'. The CJK ideographs were also used for Vietnamese until the early twentieth century.

[11] language code `mi`. Maori is usually encoded in ISO 8859-13, along with Lithuanian and Latvian!

print—all human languages, mathematical and musical notation and so on.

This is the purpose of *Unicode*[12], which allows up to 2^{32} different characters, although it is now agreed that only 2^{21} will ever be prescribed. The human languages are represented in the 'basic multilingual plane' (BMP), the first 2^{16} characters, and so most characters you will encounter have a 4-hex-digit representation. Thus `U+03B1` is alpha, `U+05D0` is aleph (the first letter of the Hebrew alphabet) and `U+30C2` is the Katakana letter 'di'.

If we all used Unicode, would everyone be happy? Unfortunately not, as sometimes the same character can be written in various ways. The Unicode standard allows 'only' 20992 slots for CJK ideographs, whereas the Taiwanese national standard defines 48711. The result is that different fonts have different variants for e.g. `U+8FCE` and a variant may be recognizable to only some of the users.

Nevertheless, Unicode is the best we have, and a decade ago Windows NT adopted the BMP of Unicode as its native encoding, using 16-bit characters. However, Unix-alike OSes expect `nul` (the zero byte) to terminate a string such as a file path, and people looked for ways to represent Unicode characters as sequences of a variable number of bytes. By far the most successful such scheme is *UTF-8*, which has over the last couple of years become the *de facto* standard for Linux distributions. So when I select

English (UK) British English

as my language at the login screen for a session in Fedora Core 3 Linux, I am also selecting UTF-8 as my encoding.

In UTF-8, the 7-bit ASCII characters are represented as a single byte. All other characters are represented as two or more bytes all with topmost bit `1`. This introduces a number of implementation issues:

- Each character is represented by a variable number of bytes, from one up to six (although it is likely at most four will ever be used).
- Some bytes can never occur in UTF-8.
- Many byte sequences are not valid in UTF-8.

The major implementation issue in internationalizing R has been to support such *multi-byte character sets* (MBCSs). There are other examples, including EUC-JP used for Japanese on Unix-alikes and the one- or two-byte encodings used for CJK on older versions of Windows.

As UTF-8 locales can support multiple languages, which are allowed for the 'legal' characters in R object names? This may depend on the OS, but in all the examples we know of all characters which would be alphanumeric in at least one human language are allowed. So for example `ñ` and `ü` are allowed in locale `en_US.utf8` on Linux, just as they were in the Latin-1 locale `en_US` (whereas on Windows the different Latin-1 locales have different sets of 'legal' characters, indeed different in different versions of Windows for some languages). We have decided that only the ASCII numbers will be regarded as numeric in R.

Sorting orders in Unicode are handled by an OS service: this is a very intricate subject so do not expect too much consistency between different implementations (which includes the same language in different encodings).

Fonts

Being able to represent alpha, aleph and the Katakana letter 'di' is one thing: being able to display them is another, and we want to be able to display them both at the console and in graphical output. Effectively all fonts available cover quite small subsets of Unicode, although they may be usable in combinations to extend the range.

R consoles normally work in monospaced fonts. That means that all printable ASCII characters have the same width. Once we move to Unicode, characters are normally categorized into three widths, one (ASCII, Greek, Arabic, ...), two (most CJK ideographs) and zero (the 'combining characters' of languages such as Thai). Unfortunately there are two conventions for CJK characters, with some fonts having e.g. Katakana single-width and some (including that used in Figure 2) double-width.

Which characters are available in a font can seem capricious: for example the one I originally used to test translations had directional single quotes but not directional double quotes.

If displaying another language is not working in a UTF-8 locale the most likely explanation is that the font used is incomplete, and you may need to install extra OS components to overcome this.

Font support for R graphics devices is equally difficult and has currently only been achieved on Windows and on comprehensively equipped X servers. In particular, `postscript()` and `pdf()` are restricted to Western and Eastern European languages as the commonly-available fonts are (and some are restricted to ISO 8859-1).

For the R user

Now that R is able to accept input in different encodings, we need a way to specify the encoding. Connections allow you to specify the encoding via the `encoding` argument, and the encoding for `stdin` can be set when reading from a file by the command-

[12] There is a more formal definition in the ISO 10646 standard, but for our purposes Unicode is a more precisely constrained version of the same encoding. There are various versions of Unicode, currently 4.0.1—see http://www.unicode.org.

line argument `--encoding`. Also, `source()` has an `encoding` argument.

It is important to make sure the encoding is correct: in particular a UTF-8 file will be valid input in most locales, but will only be interpreted properly in a UTF-8 locale or if the encoding is specified. So to read a data file produced on Windows containing the names of Swiss students in a UTF-8 locale you will need one of

```
A <- read.table(file("students",
                     encoding="latin1"))
A <- read.table(file("students",
                     encoding="UCS-2LE"))
```

the second if this is a 'Unicode' Windows file.[13]

In a UTF-8 locale, characters can be entered as e.g. `\u8fce` or `\u{8fce}` and non-printable characters will be displayed by `print()` in the first of these forms. Further, `\U` can be used with up to 8 hex digits for characters not in the BMP.

For the R programmer

There is a common assumption that one byte = one character = one display column. As from R 2.1.0 a character can use more than one byte and extend over more than one column when printed. The function `nchar()` now has a `type` argument allowing the number of bytes, characters or columns to be found—note that this defaults to bytes for backwards compatibility.

Switching between encodings

R is able to support input in different encodings by using on-the-fly translation between encodings. This should be an OS service, and for modern `glibc`-based systems it is. Even with such systems it can be frustratingly difficult to find out what encodings they support, and although most systems accept aliases such as `ISO_8859-1`, `ISO8859-1`, `ISO_8859_1`, `8859_1` and `LATIN-1`, it is quite possible to find that with three systems there is no name that they all accept. One solution is to install GNU `libiconv`: we have done so for the Windows port of R, and Apple have done so for MacOS X.

We have provided R-level support for changing encoding via the function `iconv()`, and `iconvlist()` will (if possible) list the supported encodings.

Quote symbols

ASCII has a quotation mark ", apostrophe ' and grave accent `, although some fonts[14] represent the latter two as right and left quotes respectively. Unicode provides a wide variety of quote symbols, include left and right single (U+2018, U+2019) and double (U+201C, U+201D) quotation marks. R makes use of these for `sQuote()` and `dQuote()` in UTF-8 locales.

Other languages use other conventions for quotations, for example low and mirrored versions of the right quotation mark (U+201A, U+201B), as well as guillemets (U+00AB, U+00BB; something like ≪, ≫). Looking at translations of error messages in other GNU projects shows little consistency between native writers of the same language, so we have made no attempt to define language-specific versions of `sQuote()` and `dQuote()`.

Translations

R comes with a lot of documentation in English. There is a Spanish translation of *An Introduction to R* available from CRAN, and an Italian introduction based on it as well as links to Japanese translations of *An Introduction to R*, other manuals and some help pages.

R 2.1.0 facilitates the translation of messages from 'English' into other languages: it will ship with a complete translation into Japanese and of many of the messages into Brazilian Portuguese, Chinese, German and Italian. Updates of these translations and others can be added at a later date by translation packages.

Which these 'messages' are was determined by those (mainly me) who marked up the code for possible translation. We aimed to make all but the most esoteric warning and error messages available for translation, together with informational messages such as the welcome message and the menus and dialogs of the Windows and MacOS X consoles. There are about 4200 unique messages in command-line R plus a few hundred in the GUIs.

One beneficial side-effect even for English readers has been much improvement in the consistency of the messages, as well as some re-writing where translators found the messages unclear.

For the R user

All the end user needs to do is to ensure that the desired translations are installed and that the language is set correctly. To test the latter, try it and see: very likely the internal code will be able to figure out the correct language from the locale name, and the welcome message (and menus if appropriate) will appear in the correct language. If not, try setting the environment variable `LANGUAGE` and if that does not

[13] This will probably not work unless the first two bytes are removed from the file. Windows is inclined to write a 'Unicode' file starting with a Byte Order Mark `0xFEFE`, whereas most Unix-alikes do not recognize a BOM. It would be technically easy to do so but apparently politically impossible.

[14] most likely including the one used to display this document.

help, read the appropriate section of the *R Installation and Administration* manual.

Unix-alike versions of R can be built without support for translation, and it is possible that some OSes will provide too impoverished an environment to determine the language from the locale or even for the translation support to be compiled in.

There is a pseudo-language `en@quot` for which translations are supplied. This is intended for use in UTF-8 locales, and makes use of the Unicode directional single and double quotation marks. This should be selected *via* the `LANGUAGE` environment variable.

For package writers

Any package can have messages marked for translation: see *Writing R Extensions*. Traditionally messages have been `paste`-d together, and such messages can be awkward or impossible to translate into languages with other word orders. We have added support for specifying messages *via* C-like format strings, using the R function `gettextf`. A typical usage is

```
stop(gettextf(
  "autoloader did not find '%s' in '%s'",
               name, package),
     domain = NA)
```

I chose this example as the Chinese translation reverses[15] the order of the two variables. Using `gettextf()` marks the format (the first argument) for translation and then passes the arguments to `sprintf()` for C-like formatting. The argument `domain = NA` is passed to `stop()` as the message returned by `gettextf()` will already be translated (if possible) and so needs no further translation. As the quotation marks are included in the format, translators can use other conventions (and the pseudo-language `en@quot` will).

Plurals are another source of difficulties for translators. Some languages have no plural forms, others have 'singular' and 'plural' as in English, and others have up to four forms. Even for those languages with just 'singular' and 'plural' there is the question of whether zero is singular or plural, which differs by language. There is quite general support for plurals in the R and C functions `ngettextf` and a small number[16] of examples in the R sources.

See *Writing R Extensions* for more details.

For would-be translators

Additional translations and help completing the current ones would be most welcome. Experience has shown that it is helpful to have a small translation team to cross-check each other's work, discuss idiomatic usage and to share the load. Translation will be an ongoing process as R continues to grow.

Please see the documents linked from `http://developer.r-project.org`.

Future directions

At this point we need to gain more experience with internationalization infrastructure. We know it works with the main R platforms (recent Linux, Windows, MacOS X) and Solaris and for a small set of quite diverse languages.

We currently have no way to allow Windows users to use many languages in one session, as Windows' implementation of Unicode is not UTF-8 and the standard C interface on Windows does not allow UTF-8 as an encoding. We plan on making a 'Unicode' (in the Windows sense) implementation of R 2.2.0 that uses UTF-8 internally and 16-bit characters to interface with Windows. It is likely that such a version will only be available for NT-based versions of Windows.

It is hoped to support a wider range of encodings on the `postscript()` and `pdf()` devices.

The use of encodings in documentation remains problematic, including in this article. `Texinfo` is used for the R manuals but does not currently support even ISO 8859-1 correctly. We have started with some modest support for encodings in `.Rd` files, and may be able to do more.

Acknowledgements

The work of the Japanese group (especially Ei-ji Nakama and Masafumi Okada) supplied valuable early experience in internationalization.

The translations have been made available by translation teams and in particular some enthusiastic individuals. The infrastructure for translation is provided by GNU `gettext`, suitably bent to our needs (for R is a large and complex project, and in particular extensible).

Brian D. Ripley
University of Oxford, UK
`ripley@stats.ox.ac.uk`

[15] using `'%2$s'` and `'%1$s'` in the translation to refer to the second and first argument respectively.
[16] currently 22, about 0.5% of the messages.