NS Character Set Issues:

Greg Nuyens. Aug 16,1984

This note describes a proposal for incorporating the NS character set standard into Interlisp-D. This proposal will encompass the following areas:

--implementation of a "long" character data type

It specifically omits the following:

--external file representations. **

--the reworks necessary to fonts, etc. to attain DIG, though this proposal is made with these issues in mind.

Definitions

The following terms are defined for use in the remainder of the document.

character: an instance of the "long" character data type.

byte: an 8 bit unsigned integer corresponding to the current Interlisp-D character.

character set: the (conceptual) array of 256 characters into which all the representable distinct characters have been separated. An example is character set 0 which contains most but not all of the characters currently used in Interlisp. For instance an umlaut is not included in character set 0.

font: A style in which characters are rendered. A font includes (conceptually) an image for every representable character (2^16 - 256). A font is a member of the cross product of Family, Weight, Slope, Face, Nominal Size, Rotation, and Expansion . An example is TimesRoman 12 Bold Italic Regular Compressed with 90 degree rotation. To contrast character set and font, note that only one character set (0 *?*) contains the greek letter alpha, but every font contains a rendering for that character. In practice few (no?) fonts will contain all the images of all the characters.

Overview

Before discussing how these functions will change, an overview of the scheme for the internal functions:

Error in IMAGEOBJ GETFN: SKIO.GETFN

There will be two types of character data, the previously described long and short characters. However, any single string will consist of only one type of character. Thus strings as currently represented, (effectively byte arrays) will continue to exist. However, there will also be arrays of long characters. Any functions which receive arguments of mixed type (e.g. RPLCHAR given an string of short chars and a long char to replace with) will produce results composed of long chars only.

Thus, the current implementation of substrings (as "tails" of strings) will not suffice, since the original string may be coerced upward to a long char string by an operation occurring after a substring has been returned. The plan is to use forwarding pointers in string space and store substrings as <header,offset,length> triplets. Thus when a short char string is coerced upward into a long char string, the original header will have a pointer to the new location. Any substrings returned previous to the coercion will still be valid, since they reference the (now indirect) original header. The offsets will always be scaled by a bytes-per-character value implicit in the type of string (1 for short strings, 2 for long strings).

This scheme guarantees the advertised property of substrings that they are indeed shared tails. That is, a destructive change to a substring will affect the string. This will be true regardless of any coercion of the string that occurs.

Also recognize that any coercions performed on substrings will change the representation of the whole string.

Changes to Interlisp Functions

In previous discussions the two following lists of functions affected by this proposal were identified:

First the functions which deal with the internal representation of character data:

RPLSTRING(X N Y)
: this function must change so that if either argument is "long", both are coerced to long. For the character argument, this is simply to add the default character set. For the string however, it will be necessary to copy the string coercing each character by appending the default character set.

RPLCHARCODE(X N CHARCODE)
: must now take long or short charcodes, handling them as will RPLSTRING.

GNC{CODE}(X)
: will always return an atom representing a long character (Does this mean having all the single character atoms? NO--they can be MKATOMed on the fly.) {or for GNCCODE, a long charcode}

GLC{CODE}(X)
: as above

NTHCHAR{CODE}(X N FLG RDTBL)
: will always return an atom representing a long character.

NCHARS(X FLG RDTBL)
: Returns number of characters independent of representation. When FLG is T the prin-2 length (as yet upspecified for long chars) will be used.

STRPOS{L}(PAT STRING START SKIP ANCHOR TAIL) (and MAKEBITTABLE)
: If either PAT or STRING is constructed of long chars, then the comparison will take place as though both were long. However, no destructive changes will be made.

CHARACTER(N)
: Always produces an atom whose printname is the long character whose representation is N.

CHCON(X FLG RDTBL)
: Still produces a list of charcodes. These may be short or long charcodes

SUBSTRING(X N M OLDPTR)
: Performs the substring operation, guaranteeing the EQ invariant for substrings. The internal rep'n will be the header together with the offset and length, so that if upward conversion later happens, shared tails remain.

ALLOCSTRING(N INITCHAR OLD)
: As before except, that if INITCHAR > 255 then the resulting string will be a long string. (Coercing the OLD string's char array as needed.)

MKSTRING(X FLG RDTBL)
As before, except the PName may be long.
CONCAT{LIST}(U ...) {L*}
If any of the arguments are long strings, the result is a long string.
STREQUAL
This will be true when the characters are the same, regardless of the representation.

and the following functions which must know the external representation of character data

FILEPOS
SETFILEPTR
BIN
READC
\INCHAR
\OUTCHAR
COPYBYTES
COPYCHARS

Efficiency Concerns

the common case of bytes stays fast (though some penalty)

readc is already coercing bytes upward into smallp's why not into long chars?

will it be necessary to have all 2^16 - 256 unit length pname atoms exist. (what does readc currently return)?

The intention is that the common case of short char arguments will retain their current efficiency.