Network Working Group
Internet-Draft
Intended status: Informational
Expires: June 30, 2017

A. Freytag
December 27, 2016

Variant Rules
draft-freytag-lager-variant-rules-00

Abstract

This document gives guidance on designing well-behaved Label Generation Rulesets (LGRs) that support variant labels. Typical examples of labels and LGRs are IDNs and zone registration policies defining permissible IDN labels. Variant labels are labels that are either visually or semantically indistinguishable from an applied for label and are typically delegated together with the applied-for label, or permanently reserved. While [RFC7940] defines the syntactical requirements for specifying the label generation rules for variant labels, additional considerations apply that ensure that the label generation rules are consistent and well-behaved in the presence of variants.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 30, 2017.

Table of Contents

1. Introduction

Label Generation Rulesets (LGR) [RFC7940] define permissible labels, but may also define the condition under which variant labels may exist and their status (disposition).

Successfully defining variant rules for an LGR is not trivial. A number of considerations and constraints have to be taken into account. This document describes the basic constraints and use cases for variant rules in an LGR by using a more readable notation than the XML format defined in [RFC7940]. When it comes time to capture the LGR in a formal definition, the notation used in this document can be converted to the XML format fairly directly.

From the perspective of a user of the DNS, variants are experienced as variant labels; two (or more) labels that are functionally "the same" under the conventions of the writing system used, even though their code point sequences are different. An LGR specification, on the other hand, defines variant mappings between code points, and only in a secondary step, derives the variant labels from these mappings. For a discussion of this process see [RFC7940], or as it relates to the root zone, see [Procedure].

By assigning a "type" to the variant mappings and carefully constructing the derivation of variant label dispositions from these types, the designer of an LGR can control whether some or all of the variant labels created from an original label should be available for allocation (to the original applicant) or whether some or all of these labels should be blocked instead and remain not allocatable (to anyone).

The choice of desired label disposition would be based on the expectations of the users of the particular zone, and is not the subject of this document. Instead, this document suggests how to best design an LGR to achieve the selected design choice for handling variants.

2. Variant Relationships

A variant relationship is fundamentally a "same as", in other words, it is an equivalence relationship. Now the strictest sense of "same as" would be equality, and for any equality, we have both symmetry

A = B => B = A

and transitivity

A = B and B = C => A = C

The variant relationship with its functional sense of "same as" must really satisfy the same constraint. Once we say A is the "same as" B, we also assert that B is the "same as" A. In this document, the symbol "~" means "has a variant relationship with". Thus we get

A ~ B => B ~ A

Likewise, if we make the same claim for B and C (B ~ C) then we do get A ~ C, because if B is "the same" as both A and C then A must be "the same as" C:

A ~ B and B ~ C => A ~ C

Not all relationships between labels constitute equivalence. For example, the degree to which labels are confusable is not transitive: two labels can be confusingly similar to a third without necessarily being confusable with each other, such as when the third one has a shape that is "in between" the other two. A variant relation based on (effectively) identical appearance would pass the test, as would other forms of equivalence (e.g., semantic).

3. Variant Mappings

So far, we have treated variant relationships as simple "same as" ignoring that each relationship consists of a pair of reciprocal mappings. In this document, the symbol "-->" means "maps to".

A ~ B => A --> B, B --> A

These mappings are not defined between labels, but between code points (or code point sequences). In the transitive case, given

A ~ B => A --> B, B --> A

A ~ C => A --> C, C --> A

we also get

B ~ C => B --> C, C --> B

for a total of six possible mappings. Conventionally, these are listed in tables in order of the source code point, like so

A --> B
A --> C
B --> A
B --> C
C --> A
C --> B

As we can see, each of A, B and C can be mapped two ways.

4. Variant Labels

To create a variant label, each code point in the original label is successively replaced by all variant code points defined by a mapping from the original code point. For a label AAA (the letter "A" three times), the variant labels (given the mappings from transitive example above) would be

AAB
ABA
ABB
BAA
BAB
BBA
BBB
AAC
...
CCC

5. Variant Types and Label Dispositions

Assume we wanted to allow a variant relation between some code points O and A, and perhaps also between O and B as well as O and C. By transitivity we would have

O ~ A ~ B ~ C

However, we would like to distinguish the case where someone applies for OOO from the case where someone applies for the label ABC. In the former case we would like to allocate only the label OOO, but in the latter case, we would like to also allow the allocation of either the original label OOO or the variant label ABC, or both, but not of any of the other possible variant labels, like OAO, BCO or the like. (A real-world example might be the case where O represents an unaccented letter, while A, B and C might represent various accented forms of the same letter. Because unaccented letters are a common fallback, there might be a desire to allocate an unaccented label as a variant, but not the other way around.)

How do we make that distinction?

The answer lies in labeling the mappings A --> O, B --> O, and C --> O with the type "allocatable" and the mappings O --> A, O --> B, and O --> C with the type "blocked". In this document, the symbol "x-->" means "maps with type blocked" and the symbol "a-->" means "maps with type allocatable". Thus:

O x--> A
O x--> B
O x--> C
A a--> O
B a--> O
C a--> O

When we generate all permutations of labels, we use mappings with different types depending from which code points we start.

In creating an LGR with variants, all variant mappings should always be labeled with a type ([RFC7940] does not formally require a type, but any well-behaved LGR would be fully typed). By default, these types correspond directly to the dispositions for variant labels, with the most restrictive type determining the disposition of the variant label. However, as we shall see later, it is sometimes useful to assign types from a wider array of values than the final dispositions for the labels and then define explicitly how to derive label dispositions from them.

6. Allocatable Variants

If we start with AAA, the permutation OOO will have been the result of applying the mapping A a--> O at each code point. That is, only mappings with type "a" (allocatable) were used. To know whether we can allocate both the label OOO and the original label AAA we track the types of the mappings used in generating the label.

We record the variant types for each of the variant mappings used in creating the permutation in an ordered list. Such an ordered list of variant types is called a "variant type list". In running text we often show it enclosed in square brackets. For example [a x -] means the variant label was derived from a variant mapping with the "a" variant type in the first code point position, "x" in the second code point position, and that the third position is the original code point ("-" means "no variant mapping").

For our example permutation we get the following variant type list (brackets dropped):

```
AAA --> OOO : a a a
```

From the variant type list we derive a "variant type set", denoted by curly braces, that contains an unordered set of unique variant types in the variant type list. For the variant type list for the given permutation, [a a a], the variant type set is { a }, which has a single element "a".

Deciding whether to allow the allocation of a variant label then amounts to deriving a disposition for the variant label from the variant type set created from the variant mappings that were used to create the label. For example the derivation

```
if "all variants" = "a" => set label disposition to "allocatable"
```

would allow OOO to be allocated, because the types of all variants mappings used to create that variant label from AAA are "a".

The "all-variants" condition is tolerant of an extra "-" in the variant set (unlike the "only-variants" condition described below). So, had we started with AOA, OAA or AAO, the variant set for the permuted variant OOO would have been { a - } because in each case one of the code points remains the same as the original. The "-" means that because of the absence of a mapping O --> O there is no variant type for the O in each of these labels.

The "all-variants" = "a" condition ignores the "-", so using the derivation from above, we find that OOO is an allocatable variant for each of the labels AOA, OAA or AAO.

7. Blocked Variants

Blocked variants are not available to another registrant. They therefore protect the applicant of the original label from someone else registering a label that is "the same as" under some user-perceived metric. Blocked variants can be a useful tool even for scripts for which no allocatable labels are ever defined.

If we start with OOO, the permutation AAA will have been the result of applying only mappings with type "blocked" and we cannot allocate the label AAA, only the original label OOO. This corresponds to the following derivation:

```
if "any variants" = "x" => set label disposition to "blocked"
```

To additionally prevent allocating ABO as a variant label for AAA we further need to make sure that the mapping A --> B has been defined with type "blocked" as in

```
A  x--> B
```

so that

```
AAA --> ABO: - x a.
```

Thus the set {x a} contains at least one "x" and satisfies the derivation of a blocked disposition for ABO when AAA is applied for.

8. Pure Variant Labels

Now, if we wanted to prevent allocation of AOA when we start from AAA, we would need a rule disallowing a mix of original code points and variant code points, which is easily accomplished by use of the "only-variants" qualifier, which requires that the label consist entirely of variants and all the variants are from the same set of types.

if "only-variants" = "a" => set label disposition to "allocatable"

The two code points A in AOA are not arrived at by variant mappings, because the code points are unchanged and no variant mappings are defined for A --> A. So, in our example, the set of variant mapping types is

AAA --> AOA: - a -

but unlike the "all-variants" condition, "only-variants" requires a variant type set { a } corresponding to a variant type list [a a a] (no - allowed). By adding a final derivation

else if "any-variants" = "a" => set label disposition to "blocked"

and executing that derivation only on any remaining labels, we disallow AOA when starting from AAA, but still allow OOO.

Derivation conditions are always applied in order, with later derivations only applying to labels that did not match any earlier conditions, as indicated by the use of "else" in the last example. In other words, they form a cascade.

9. Reflexive Variants

But what if we started from AOA? We would expect OOO to be allocatable, but the variant type set would be

OOO --> OOO: a - a

because the O is the original code point. Here is where we use a reflexive mapping, by realizing that O is "the same as" O, which is normally redundant, but allows us to specify a disposition on the mapping

O a--> O

with that, the variant type list for OOO --> OOO becomes:

AOA --> OOO: a a a

and the label OOO again passes the derivation condition

if "only-variants" = "a" => set label disposition to "allocatable"

as desired. This use of reflexive variants is typical whenever derivations with the "only-variants" qualifier are used. If any code

point uses a reflexive variant, a well-behaved LGR would specify an appropriate reflexive variant for all code points.

10. Limiting Allocatable Variants by Subtyping

As we have seen, the number of variant labels can potentially be large, due to combinatorics.

To recap, in the standard case a code point C can have (up to) two types of variant mappings

```
C x--> X
C a--> A
```

where a--> means a variant mapping with type "allocatable", and x--> means "blocked". For the purpose of this discussion, we name the target code point with the corresponding uppercase letter.

Subtyping is a mechanism that allows us to distinguish among different types of allocatable variants. For example, we can define three new types: "s", "t" and "b". "s" and "t" are mutually incompatible, but "b" is compatible with either "s" or "t" (in this case, "b" stands for "both"). With this, a code point C might have (up to) four types of variant mappings

```
C  x--> X
C  s--> S
C  t--> T
C  b--> B
```

and explicit reflexive mappings of one of these types

```
C  s--> C
C  t--> C
C  b--> C
```

As before, all mappings must have one and only one type, but each code point may map to any number of other code points.

We define the compatibility of "b" with "t" or "s" by our choice of derivation conditions as follows

```
if "only-variants" = "s" or "b" =>  allocatable
else if "only-variants" = "t" or "b" => allocatable
else if "any-variants" = "s" or "t" or "b"  or "x" => blocked
```

An original label of four code points

CCCC

may have many variant labels such as this example listed with its corresponding variant type list:

CCCC --> XSTB : x s t b

This variant label is blocked because to get from C to B required x-->. (Because variant mappings are defined for specific source code points, we need to show the starting label for each of these examples, not merely the code points in the variant label.) . The variant label

CCCC --> SSBB : s s b b

is allocatable, because the variant type list contains only allocatable mappings of subtype s or b, which we have defined as being compatible by our choice of derivations. The actual set of variant types {s, b} has only two members, but the examples are easier to follow if we list each type. The label

CCCC --> TTBB : t t b b

is again allocatable, because the variant type set {t, b} contains only allocatable mappings of the mutually compatible allocatable subtypes t or b. In contrast,

CCCC --> SSTT : s s t t

is not allocatable, because the type set contains incompatible subtypes t and s and thus would be blocked by the final derivation.

The variant labels

CCCC --> CSBB : c s b b
CCCC --> CTBB : c t b b

are only allocatable based on the subtype for the C --> C mapping, which is denoted here by c and (depending on what was chosen for the type of the reflexive mapping) could correspond to s, t, or b.

If it is s, the first of these two labels is allocatable; if it is t, the second of these two labels is allocatable; if it is b, both labels are allocatable.

So far, the scheme doesn't seem to have brought any huge reduction in allocatable variant labels, but that is because we tacitly assumed

that C could have all three types of allocatable variants s, t, and b at the same time.

In a real world example, the types s, t and b are assigned so that each code point C normally has at most one non-reflexive variant mapping labeled with one of these subtypes, and all other mappings would be assigned type x (blocked). This holds true for most code points in existing tables (such as those used in current IDN TLDs), although certain code points have exceptionally complex variant relations and may have an extra mapping.

11. Allowing Mixed Originals

If the desire is to allow original labels (but not variant labels) that are s/t mixed, then the scheme needs to be slightly refined to distinguish between reflexive and non-reflexive variants. In this document, the symbol "r-n" means "a reflexive (identity) mapping of type 'n'". The reflexive mappings of the preceding section thus become:

```
C r-s--> C
C r-t--> C
C r-b--> C
```

With this convention, and redefining the derivations

```
if "only-variants" = "s"  or "r-s" or "b" or "r-b" => allocatable
else if "only-variants" = "t" or "r-t" or  "b" or "r-b" => allocatable
else if "any-variants" = "s" or "t" or "b"  or "x" => blocked
else => allocatable
```

any labels that contain only reflexive mappings of otherwise mixed type (in other words, any mixed original label) now fall through and their disposition is set to "allocatable" in the final derivation.

12. Handling Out Of Repertoire Variants

At first it may seem counterintuitive to define variants that map to code points not part of the repertoire. However, for zones for which multiple LGRs are defined, there may be situations where labels valid under one LGR should be blocked if a label under another LGR is already delegated. This situation can arise whether or not the repertoires of the affected LGRs overlap, and, where repertoires overlap, whether or not the labels are both restricted to the common subset.

In order to handle this exclusion relation through definition of variants, it is necessary to be able to specify variant mappings to some code point X that is outside an LGR's repertoire, R:

```
C  x--> X : where C = elementOf(R) and X != elementOf(R)
```

Because of symmetry, it is necessary to also specify the inverse mapping in the LGR:

```
X x--> C : where X != elementOf( R) and C = elementOf( R)
```

This makes X a source of variant mappings and it becomes necessary to identify X as being outside the repertoire, so that any attempt to apply for a label containing X will lead to a disposition of "invalid" - just as if X had never been listed in the LGR. The mechanism to do this, again uses reflexive variants, but with a new type of reflexive mapping of "out-of-repertoire-var", shown as "r-o-->":

```
X r-o--> X
```

When paired with a suitable derivation, any label containing X is assigned a disposition of "invalid", just as if X was any other code point not part of the repertoire. The derivation used is:

```
if "any-variant" = "out-of-repertoire-var" => invalid
```

It is inserted ahead of any other derivation of the "any-variant" kind in the chain of derivations. As a result for any out-of repertoire variants three entries are minimally required:

```
C  x--> X : where C = elementOf( R) and X != elementOf( R)
X  x--> C : where X = !elementOf( R) and C = elementOf( R)
X r-o--> X : where X = !elementOf( R)
```

Because no variant label with any code point outside the repertoire could ever be allocated, the only logical choice for the non-reflexive mappings to out-of-repertoire code points is "blocked".

13. Conditional Variants

Variant mappings are based on whether code points are "the same" to the user. In some writing systems, code points change shape based on where they occur in the word (positional forms). Some code points have matching shapes in some positions, but not in others. In such cases, the variant mapping only exists for some possible positions, or more general, only for some contexts. For other contexts, the variant mapping does not exist.

For example, take two code points that have the same shape at the end of a label (or in final position) but not in any other position. In that case, they are variants only when they occur in the final position, something we indicate like this:

```
final: C --> D
```

In cursively connected scripts, like Arabic, a code point may take its final form when next to any following code point that interrupts the cursive connection, not just at the end of a label. (We ignore the isolated form to keep the discussion simple, if it was included, "final" might be "final-or-isolate", for example).

From symmetry, we expect that the mapping D --> C should also exist only when the code point D is in final position. (Similar considerations apply to transitivity).

Sometimes a code point has a final form that is practically the same as that of some code point while sharing initial and medial forms with another.

```
final: C --> D
!final: C --> E
```

Here the case where the condition is the opposite of final is shown as "!final".

Because shapes differ by position, when a context is applied to a variant mapping, it is treated independently from the same mapping in other contexts. This extends to the assignment of types. For example, the mapping C --> F may be "allocatable" in final position, but "blocked" in any other context:

```
final:  C a--> F
!final: C x--> F
```

Now, the type assigned to the forward mapping is independent of the reverse symmetric mapping, or any transitive mappings. Imagine a situation where the symmetric mapping is defined as F a--> C, that is, all mappings from F to C are "allocatable":

```
final: F a--> C
!final: F a-->C
```

Why not simply write F a--> C? Because the forward mapping is divided by context. Adding a context makes the two forward variant mappings distinct and that needs to be accounted for explicitly in the reverse mappings so that human and machine readers can easily

verify symmetry and transitivity of the variant mappings in the LGR. (This is true even though the two opposite contexts "final" and "!final" should together cover all possible cases).

14. Conditional Variants and Well-Behaved LGRs

A well-behaved LGR with contextual variants always uses "fully qualified" variant mappings and always agrees in the names of the context rules for forward and reverse mappings. It also ensures that no label can match more than one context for the same mapping. Using mutually exclusive contexts, such as "final" and "!final" is an easy way to ensure that.

However, it is not always necessary to define dual or multiple contexts that together cover all possible cases. For example, here are two contexts that do not cover all possible positional contexts:

```
final: C --> D
initial: C --> D.
```

A well-behaved LGR using these two contexts, would define all symmetric and transitive mappings involving C, D and their variants consistently in terms of the two conditions "final" and "initial" and ensure both cannot be satisfied at the same time by some label.

In addition to never defining the same mapping with two contexts that may be satisfied by the same label, a well-behaved LGR never combines a variant mapping with context with the same variant mapping without a context:

```
context: C --> D
C --> D
```

Inadvertent mixing of conditional and unconditional variants can be detected and flagged by a parser, but verifying that two formally distinct contexts are never satisfied by the same label would depend on the interaction between labels and context rules, which means that it will be up to the LGR designer to ensure the LGR is well-behaved.

A well-behaved LGR never assigns conditions on a reflexive variant, as that is effectively no different from having a context on the code point itself; the latter is preferred.

Finally, for symmetry to work as expected, the context must be defined such that it is satisfied for both the original code point in the context of the original label and for the variant code point in the variant label. In other words the context should be "stable under variant substitution" anywhere in the label.

Positional contexts usually satisfy this last condition; for example, a code point that interrupts a cursive connection would likely share this property with any of its variants. However, as it is in principle possible to define other kinds of contexts, it is necessary to make sure that the LGR is well behaved in this aspect at the time the LGR is designed.

In summary, conditional contexts can be an essential tool, but some additional care must be taken to ensure that an LGR containing conditional contexts is well behaved.

15. Variants for Sequences

Variants mappings can be defined between sequences, or between a code point and a sequence. Such variants are no different from variants defined between single code points, except if a sequence is defined such that there is a code point or shorter sequence that is a prefix (initial subsequence) and the remainder is also part of the repertoire. In that case, it is possible to create duplicate variants with conflicting dispositions.

The following shows such an example resulting in conflicting reflexive variants:

```
A a--> C
AB x--> CD
```

where AB is a sequence with an initial subsequence of A. For example, B might be a combining code point used in sequence AB. If B only occurs in the sequence, there is no issue, but if B also occurs by itself, for example:

```
B a--> D
```

then a label "AB" might correspond to either {A}{B}, that is the two code points, or {AB}, the sequence, where the curly braces show the sequence boundaries as they would be applied during label validation and variant mapping.

A label AB would then generate the "allocatable" variant label {C}{D} and the "blocked" variant label {CD} thus creating two variant labels with conflicting dispositions.

The easiest way to avoid an ambiguous segmentation into sequences is by never allowing both a sequence and all of its constituent parts simultaneously as independent parts of the repertoire, for example, by not defining B.

Sequences are often used for combining sequences, and not allowing the combining mark by itself prevents it from occurring outside of specifically enumerated contexts. In cases where this cannot be done, other techniques can be used to prevent ambiguous segmentation, for example, a context rule on code points that would disallow A preceding B in any label except as part of a predefined sequence. The details of such techniques are outside the scope of this document (see [RFC7940] for information on context rules for code points).

16. Corresponding XML Notation

The XML format defined in [RFC7940] corresponds fairly directly to the notation used in this document. For example, a variant relation of type "blocked"

```
C  x--> X
```

is expressed as

```
<char cp="nnnn">
  <var cp="mmmm" type="blocked" />
</char>
```

where we assume that nnnn and mmmm are the [Unicode9] code point values for C and X, respectively. A reflexive mapping always uses the same code point value for <char> and <var> element, for example

```
X r-o--> X
```

would correspond to

```
<char cp="nnnn"><var cp="nnnn" type="out-of-repertoire-var" /></char>
```

Multiple <var> elements may be nested inside a single <char> element, but their "cp" values must be distinct (unless other distinguishing attributes are present that are not discussed here).

```
<char cp="nnnn">
  <var cp="kkkk" type="allocatable" />
  <var cp="mmmm" type="blocked" />
</char>
```

A set of conditional variants like

```
final: C a--> K
!final: C b--> K
```

would correspond to

```
<var cp="kkkk" when="final" type="allocatable" />
<var cp="kkkk" not-when="final" type="blocked" />
```

where the string "final" references a name of a context rule. Context rules are defined in [RFC7940] and the details of how to create and define them are outside the scope of this document. If the label matches the context defined in the rule, the variant mapping is valid and takes part in further processing. Otherwise it is invalid and ignored. Using the "not-when" attribute inverts the sense of the match. The two attributes are mutually exclusive.

A derivation of a variant label disposition

```
if "only-variants" = "s" or "b" => allocatable
```

is expressed as

```
<action disp="allocatable" only-variants= "s b" />
```

Instead of using "if" and "else if" the <action> elements implicitly form a cascade, where the first action triggered defines the disposition of the label. The order of action elements is thus significant.

For the full specification of the XML format see [RFC7940].

17. IANA Considerations

This document does not specify any IANA actions.

18. Security Considerations

There are no security considerations for this memo.

19. References

19.1. Normative References

[RFC7940] Davies, K. and A. Freytag, "Representing Label Generation Rulesets Using XML", RFC 7940, DOI 10.17487/RFC7940, August 2016, <http://www.rfc-editor.org/info/rfc7940>.

19.2. Informative References

[Procedure]
Internet Corporation for Assigned Names and Numbers, "Procedure to Develop and Maintain the Label Generation Rules for the Root Zone in Respect of IDNA Labels", 2013, <http://www.icann.org/en/resources/idn/variant-tlds/draft-lgr-procedure-20mar13-en.pdf>.

[Unicode9]
The Unicode Consortium, "The Unicode Standard, Version 9.0.0", ISBN 978-1-936213-13-9, 2016, <http://www.unicode.org/versions/Unicode9.0.0/>.

Preferred Citation: The Unicode Consortium. The Unicode Standard, Version 9.0.0, (Mountain View, CA: The Unicode Consortium, 2016. ISBN 978-1-936213-13-9)

Appendix A. Acknowledgements

Contributions that have shaped this document have been provided by Marc Blanchet, Sarmad Hussain, Nicholas Ostler, Michel Suignard, and Wil Tan.

Appendix B. Change Log

RFC Editor: Please remove this appendix before publication.

-00 Initial draft.

Author's Address

Asmus Freytag

Email: asmus@unicode.org