Notation

This section discusses notation used in this specification.

All operations in this document are implicitly performed modulo 2^{32}. We use standard mathematical notation for addition, subtraction, multiplication, and exponentiation. Division always denotes integer division, i.e. any remainder is dropped.

We also use the following operators:

- $x \ll y$ denotes shifting x to the left y bits, i.e. $x \ll y = x2^y$
- $x \gg y$ denotes a *logical* right shift – it shifts x to the right by y bits, i.e. $x \gg y = \frac{x}{2^y}$
- $\text{ROTL}^n(x)$ rotates x n bits to the left, i.e. $\text{ROTL}^n(x) = (x \ll n) + (x \gg (32 - n))$

The RRS Rolling Checksums

The `rrs` family of checksums were first used in rsync, and later in bup and perkeep. `rrs` was originally inspired by the adler-32 checksum. The name `rrs` was chosen for this specification, and stands for `rsync rolling sum`.

Definition

A concrete `rrs` checksum is defined by the parameters:

- M, the modulus
- C, the character offset
- R, the rotation.

Given a sequence of bytes $X_0, X_1, ..., X_N$ and a choice of M and C, the `rrs` hash of the sub-sequence $X_k, ..., X_l$ is $s(k, l)$, where:

$a(k, l) = (\sum_{i=k}^{l}(X_i + C)) \mod M$

$b(k, l) = (\sum_{i=k}^{l}(l - i + 1)(X_i + C)) \mod M$

$s(k, l) = \text{ROTL}^R(a(k, l) + 2^{16}b(k, l))$

RRS0

The concrete hash called `rrs0` uses the values:

- $M = 2^{16}$
- $C = 31$
- $R = 0$

`rrs0` is used by current versions of librsync as of August 2020. Note that the hash in the rsync documentation is not `rrs0`; that hash uses $C = 0$.

RRS1

The concrete hash called `rrs1` uses the values:

- $M = 2^{16}$
- $C = 31$
- $R = 16$

`rrs1` is used by both Bup and Perkeep, and implemented by the go package `go4.org/rollsum`.

Implementation

Rolling

`rrs` is a family of *rolling* hashes. We can compute hashes in a rolling fashion by taking advantage of the fact that:

$$a(k+1, l+1) = (a(k, l) - (X_k + C) + (X_{l+1} + C)) \mod M$$

$$b(k+1, l+1) = (b(k, l) - (l - k + 1)(X_k + C) + a(k+1, l+1)) \mod M$$

So, a typical implementation will work like:

- Keep $X_k, ..., X_l$ in a ring buffer.
- Also store $a(k, l)$ and $b(k, l)$.
- When X_{l+1} is added to the hash:
  - Dequeue X_k from the ring buffer, and enqueue X_{l+1}.
  - Use X_k, X_{l+1}, and the stored $a(k, l)$ and $b(k, l)$ to compute $a(k+1, l+1)$ and $b(k+1, l+1)$. Then use those values to compute $s(k+1, l+1)$ and also store them for future use.

Choice of M

Choosing $M = 2^{16}$ has the advantages of simplicity and efficiency, as it allows $s(k, l)$ to be computed using only shifts and bitwise operators; in C:

```c
uint32_t s = a | (b << 16);
```