Thinning Thinning

Fast and Safe Bits and Bobs for Type Checkers

by April Gonçalves and Wen Kokke

Let's say you're building a type checker...

```haskell
type Ix = Int

data Tm where
  Var :: Ix → Tm
  Lam :: Tm → Tm
  App :: Tm → Tm → Tm
```

```haskell
-- you, building a type checker:
--
-- ヽ(ˆ▽ˆ)ノ — "yay, I love me some
--             nameless representation!"

-- you, ten minutes later:
--
-- "why the numbers bad?" — (˚ ˃̣̣̥⌓˂̣̣̥)
```

```haskell
-- ┌ example terms go here
-- ↓
idTm    = Lam (Var 0)
constTm = Lam (Lam (Var 1))
```

```haskell
eval :: Tm → Tm
eval = ???

subst :: (Ix → Tm) → Tm → Tm
subst = ???
```

Let's say you're building a type checker...

```haskell
type Ix = Int

data Tm where
  Var :: Ix → Tm
  Lam :: Tm → Tm
  App :: Tm → Tm → Tm
```

```haskell
-- you, building a type checker:
--
-- ¯\(ツ)/¯ — "yay, I love me some
--            nameless representation!"

-- you, ten minutes later:
--
-- "why the numbers bad?" — (ಠ_ಠ)
```

```haskell
-- ┌ example terms go here
-- ↓
idTm    = Lam (Var 0)
constTm = Lam (Lam (Var 1))
```

```haskell
eval :: Tm → Tm
eval = ???

subst :: (Ix → Tm) → Tm → Tm
subst = ???
```

Let's say you're building a type checker...

```
type data N = Z | S N

data Ix (n :: N) where
  Z ::          Ix (S n)
  S :: Ix n -> Ix (S n)

data Tm (n :: N) where
  Var :: Ix n
  Lam :: Tm (S n) -> Tm n
  App :: Tm n -> Tm n -> Tm n

idTm    = Lam (Var Z)
constTm = Lam (Lam (Var (S Z)))
```

```
-- you, building a type checker:
--
-- ヽ(ᵔᗜᵔ)ﾉ — "yay, I love me some
--      well-scoped representation!"

-- you, ten minutes later:
--
-- "why it run so slow?" — (˚ ˃̣̣̥⌓˂̣̣̥)

eval :: Tm n -> Tm n
eval = aww_yeah_its_easy

subst :: Env n m -> Tm n -> Tm m
subst = just_do_what_type_says
```

Let's say you're building a type checker...

```
type data N = Z | S N

data Ix (n :: N) where
  Z ::          Ix (S n)
  S :: Ix n -> Ix (S n)

data Tm (n :: N) where
  Var :: Ix n
  Lam :: Tm (S n) -> Tm n
  App :: Tm n -> Tm n -> Tm n

idTm    = Lam (Var Z)
constTm = Lam (Lam (Var (S Z)))
```

```
-- you, building a type checker:
--
-- ¯\(ツ)/¯ — "yay, I love me some
--      well-scoped representation!"

-- you, ten minutes later:
--
-- "why it run so slow?" — (ノ ಠ益ಠ)

eval :: Tm n -> Tm n
eval = aww_yeah_its_easy

subst :: Env n m -> Tm n -> Tm m
subst = just_do_what_type_says
```

Do you ever wish you could have fast and safe?

Now, with the power of pattern synonyms, view patterns, and lies, you can!

```
newtype Ix (n :: N) = UnsafeIx Int

-- construct `Z`
mkZ = coerce 0 :: Ix (S n)

-- construct `S i` from `i`
mkS :: Ix n → Ix (S n)
mkS = coerce (+1)

-- destruct `S i` into `i`
unS :: Ix n → Ix (P n)
unS = coerce (-1)
```

```
-- ← that's gotta be a nominal role
--    but let's coerce as a shortcut

-- eliminate an `Ix n` into an `a`
el :: a → (Ix (P n) → a) → Ix n → a
el z s i =
  if i == mkZ then z else s (unS i)

--  ┌ type-level predecessor is here
-- ↓
type family P :: N → N where
  P (S n) = n
```

Do you ever wish you could have fast and safe?

Now, with the power of pattern synonyms, view patterns, and lies, you can!

```
-- the base functor for the safe Ix
data IxF (ix :: N → *) (n :: N) where
  ZF ::          IxF (S n)
  SF :: ix n → IxF (S n)

prj :: Ix n → IxF Ix n
prj = el (uc ZF) (uc SF)
  where uc = unsafeCoerce

emb :: IxF Ix n → Ix n
emb ZF      = mkZ
emb (SF i) = mkS i
```

```
-- so `Pos n` means `P n` exists
type Pos (n :: Nat) = n ~ S (P n)

pattern Z :: (Pos n) ⇒ Ix n
pattern Z ← (prj → ZF)
  where Z = emb ZF

pattern S :: (Pos n) ⇒ Ix (P n) → Ix n
pattern S i ← (prj → ZS i)
  where S i = emb (SF i)

-- …and we have constructors!
```

Do you ever wish you could have fast and safe?

Now, with the power of pattern synonyms, view patterns, and lies, you can!

```
-- …which are just like the safe Ix    -- ← except they take `2` words in
-- constructors we started out with!   --    memory instead of `2*n` words

thin ∷ Ix (S n) → Ix n → Ix (S n)      --
thin    Z     j  = S j                 -- ← that's just the old linear time
thin (S _)    Z  = Z                   --    function! boo! we can do better
thin (S i) (S j) = S (thin i j)        --

-- should you? no! make it go fast!
thin ∷ Ix (S n) → Ix n → Ix (S n)      --
thin = coerce $ \i j →                 -- ← that's constant time, babeeeee!
  if i ≤ j then S j else j             --
```

Wasn't this talk about thinning thinning? Great segue, me!

```
-- a thinning `n ≤ m` tells you how
-- you get from stuff with `m` things
-- to stuff with `n` things in scope.

data (≤) (n ∷ N) (m ∷ N) where
  Refl ∷ n ≤ n
  Keep ∷ n ≤ m → S n ≤ S m
  Drop ∷ n ≤ m →   n ≤ S m

--    1     2     3     4     1     2
nm = Keep (Drop (Keep (Drop  Refl)))
--    0     1     0     1     0     0
--    ↓     ↓     ↓     ↓     ↓     ↓
--    1     _     3     _     1     2
```

```
-- let's use the same technique!

newtype (≤) (n ∷ N) (m ∷ N) =
  UnsafeTh Word

-- a thinning is a bit vector
-- * `Refl` is all `0` bits
-- * `Keep` adds `0` onto the start
-- * `Drop` adds `1` onto the start
mkRefl    = 0
mkKeep nm = nm `shift` 1
mkDrop nm = nm `shift` 1 .|. 1


-- not pictured: everything else!
```

Wasn't this talk about thinning thinning? Great segue, me!

```
-- a thinning `n ≤ m` tells you how
-- you get from stuff with `m` things
-- to stuff with `n` things in scope.

data (≤) (n :: N) (m :: N) where
  Refl :: n ≤ n
  Keep :: n ≤ m → S n ≤ S m
  Drop :: n ≤ m →   n ≤ S m

--   1     2     3     4     1     2
nm = Keep (Drop (Keep (Drop  Refl)))
--   0     1     0     1     0     0
--   ↓     ↓     ↓     ↓     ↓     ↓
--   1     _     3     _     1     2
```

```
-- let's use the same technique!

newtype (≤) (n :: N) (m :: N) =
  UnsafeTh Word

-- a thinning is a bit vector
-- * `Refl` is all `0` bits
-- * `Keep` adds `0` onto the start
-- * `Drop` adds `1` onto the start
mkRefl    = 0
mkKeep nm = nm `shift` 1
mkDrop nm = nm `shift` 1 .|. 1

-- not pictured: everything else!
```

Wasn't this talk about thinning thinning? Great segue, me!

```
-- a thinning `n ≤ m` tells you how
-- you get from stuff with `m` things
-- to stuff with `n` things in scope.

data (≤) (n :: N) (m :: N) where
  Refl :: n ≤ n
  Keep :: n ≤ m → S n ≤ S m
  Drop :: n ≤ m →   n ≤ S m

--   1     2     3     4     1     2
nm = Keep (Drop (Keep (Drop  Refl)))
--   0     1     0     1     0     0
--   ↓     ↓     ↓     ↓     ↓     ↓
--   1     _     3     _     1     2
```

```
-- let's use the same technique!

newtype (≤) (n :: N) (m :: N) =
  UnsafeTh Word

-- a thinning is a bit vector
-- * `Refl` is all `0` bits
-- * `Keep` adds `0` onto the start
-- * `Drop` adds `1` onto the start
mkRefl    = 0
mkKeep nm = nm `shift` 1
mkDrop nm = nm `shift` 1 .|. 1

-- not pictured: everything else!
```

Wasn't this talk about thinning thinning? Great segue, me!

```
-- a thinning `n ≤ m` tells you how
-- you get from stuff with `m` things
-- to stuff with `n` things in scope.

data (≤) (n :: N) (m :: N) where
  Refl :: n ≤ n
  Keep :: n ≤ m → S n ≤ S m
  Drop :: n ≤ m →   n ≤ S m

--   1     2     3     4     1     2
nm = Keep (Drop (Keep (Drop  Refl)))
--   0     1     0     1     0     0
--   ↓     ↓     ↓     ↓     ↓     ↓
--   1     _     3     _     1     2
```

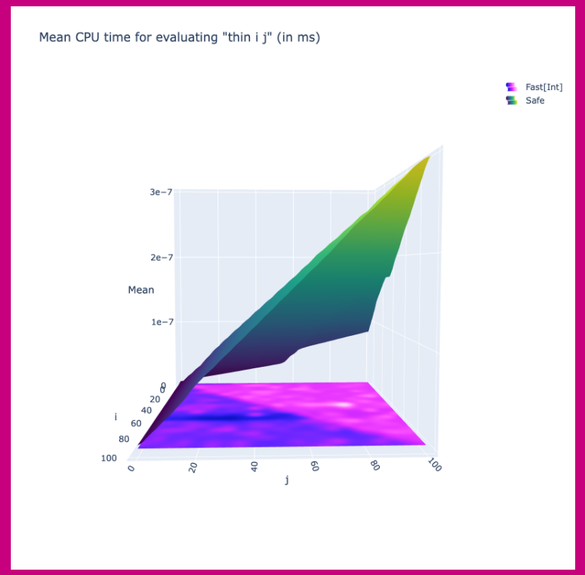
```
-- let's use the same technique!

newtype (≤) (n :: N) (m :: N) =
  UnsafeTh Word

-- a thinning is a bit vector
-- * `Refl` is all `0` bits
-- * `Keep` adds `0` onto the start
-- * `Drop` adds `1` onto the start
mkRefl     = 0
mkKeep nm = nm `shift` 1
mkDrop nm = nm `shift` 1 .|. 1

-- not pictured: everything else!
```

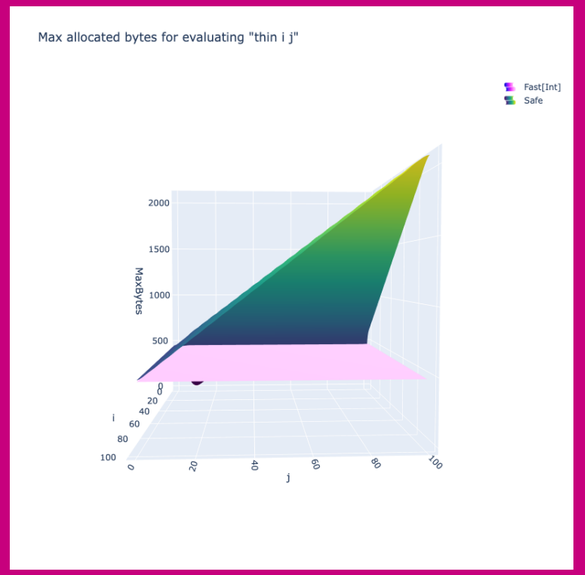
We've got thin thinnings! Let's thin thinning thinning!

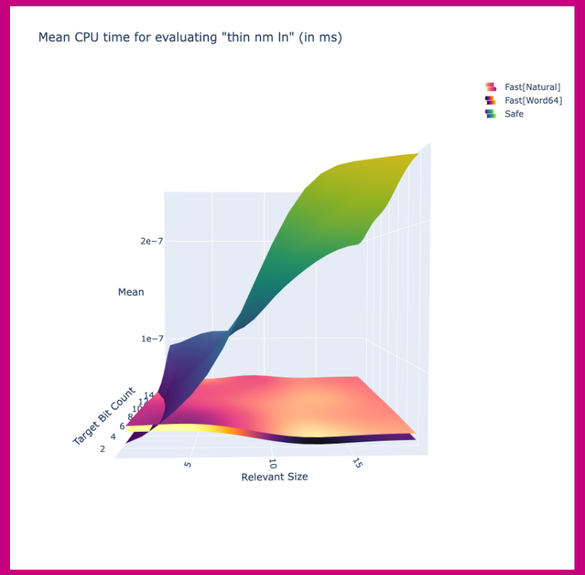
```
-- these are just the constructors of safe thinnings we started out with!
-- here's thinning thinnings — or thinning composition — to prove it!

thinThin :: n ≤ m → l ≤ n → l ≤ m
thinThin nm         Refl       = nm
thinThin Refl       ln         = ln
thinThin (Keep nm) (Keep ln) = Keep (thinThin nm ln)
thinThin (Keep nm) (Drop ln) = Drop (thinThin nm ln)
thinThin (Drop nm) ln         = Drop (thinThin nm ln)

-- have we learned our lesson? apparently not. make it faster!
thinThin = coerce $ \nm ln → nm .|. (pdep ln (complement nm))
--                                    ↑
-- that's one single x86 instruction ┘ that's 3 instructions total, babeeeee!
```

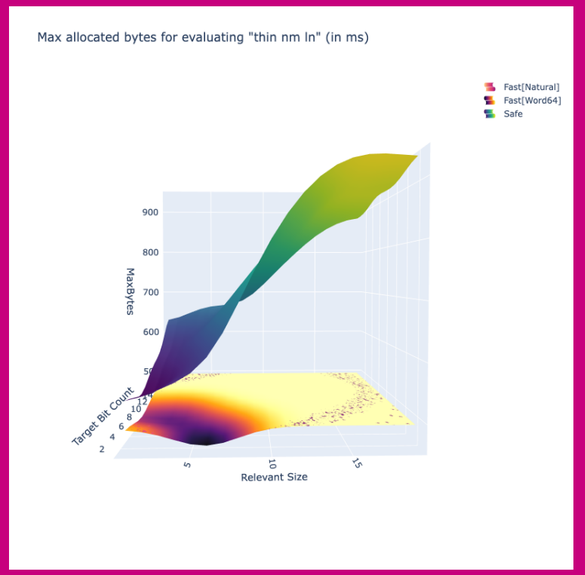
Wrapping up. Let's do a speed run.

- Released on Hackage as data-debruijn
- Is it safe? QuickCheck says yes. Every fast function, constructor, and pattern is checked against the safe version.
- Is it fast? I say yes. Have some graphs.

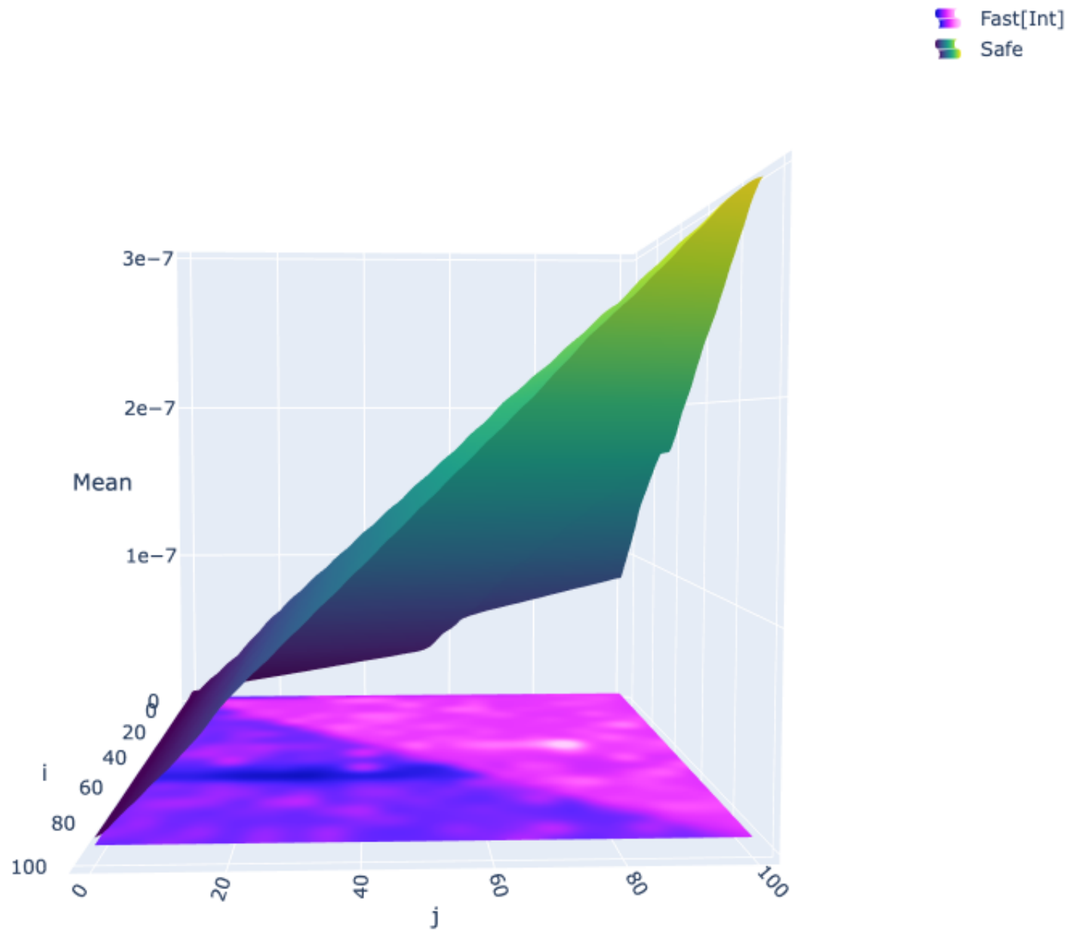




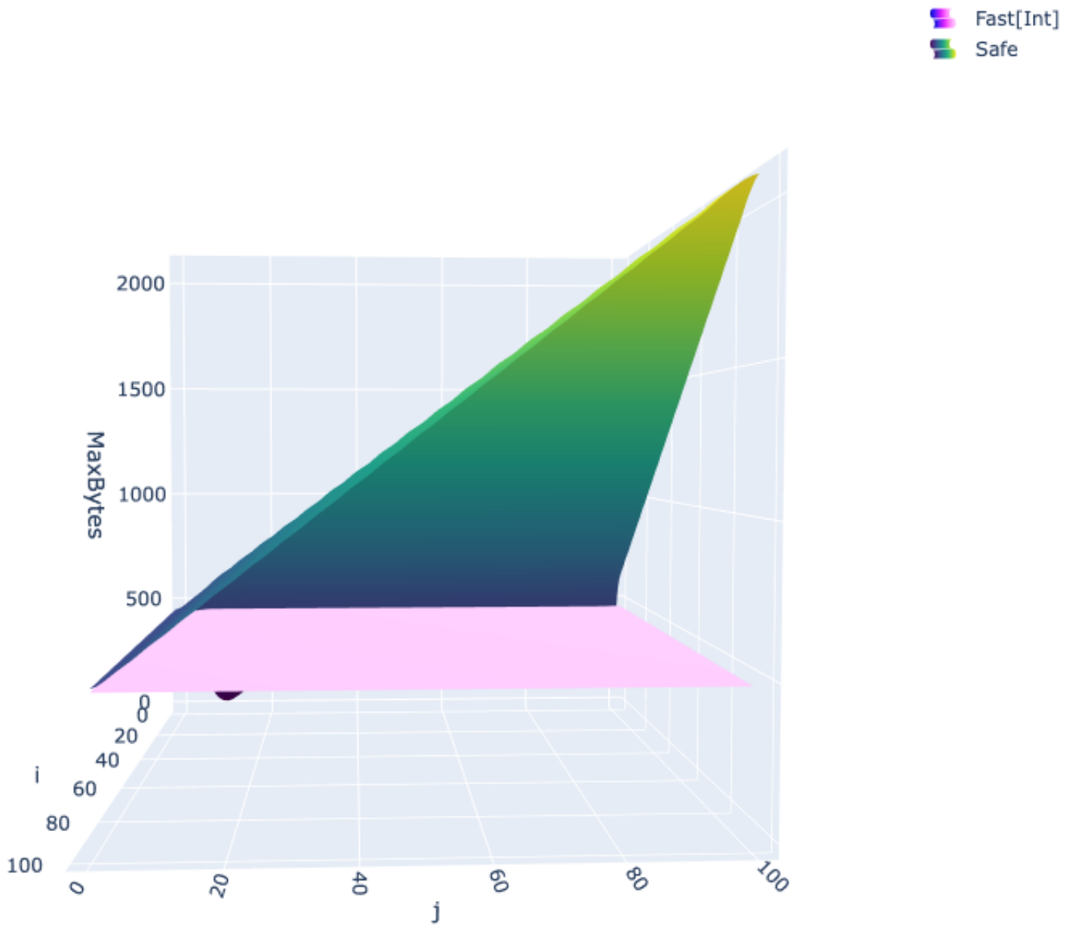




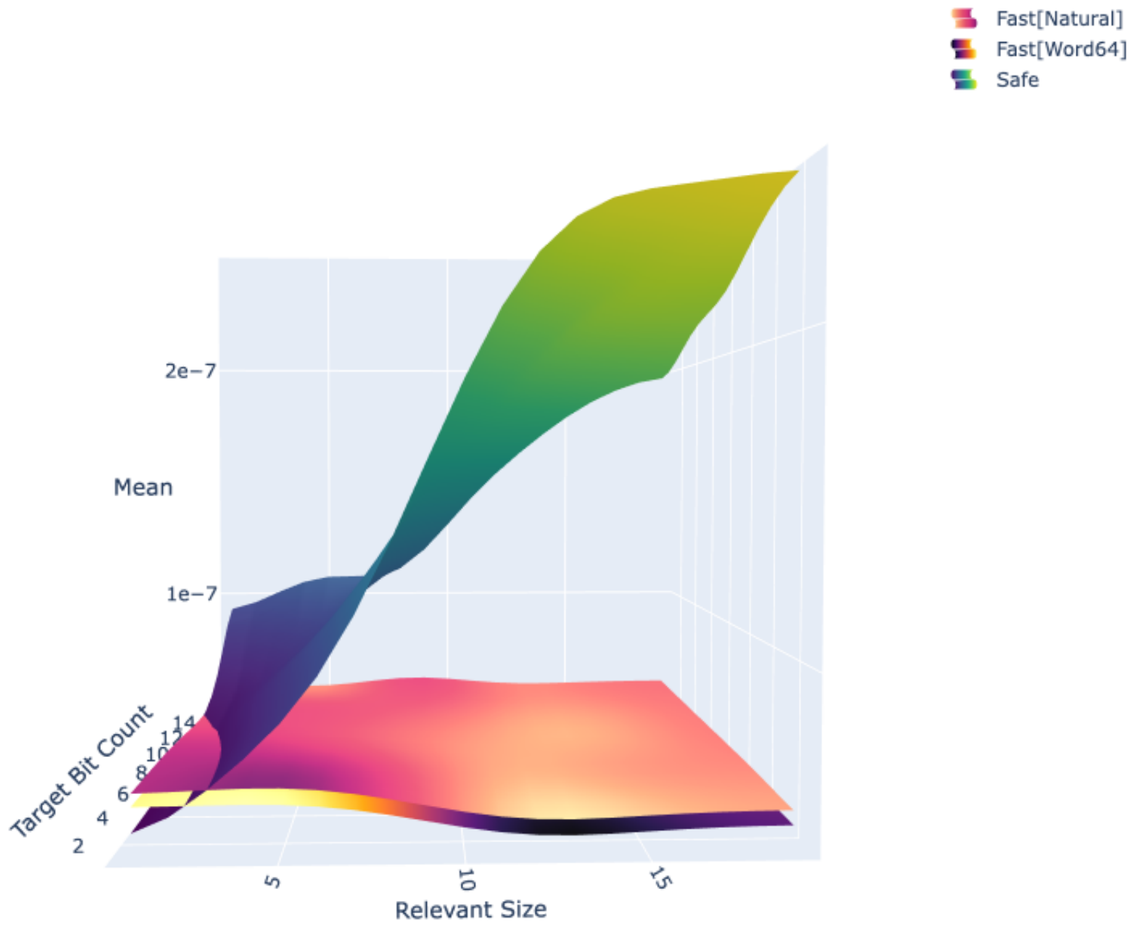
Mean CPU time for evaluating "thin i j" (in ms)



Max allocated bytes for evaluating "thin i j"



Mean CPU time for evaluating "thin nm ln" (in ms)



Max allocated bytes for evaluating "thin nm ln" (in ms)