Making Sense of Tristate Numbers (tnum)

Shung-Hsi Yu
SUSE

LINUX PLUMBERS CONFERENCE Vienna, Austria / Sept. 18-20, 2024

What is **tnum?**

Tristate

Number

Tracked

```
struct bpf_reg_state {
  enum bpf_reg_type type;
  struct tnum {
    u64 value;
    u64 mask;
  } var_off;
  ...
};
```

kernel/bpf/tnum.c

Number of **bugs** in kernel/bpf/tnum.c

since its introduction in 2017

1

Number of **lines** in kernel/bpf/tnum.c

remains unchanged since 2017

168

out of 213

75%

Good **code**

Good **API**

In practice…

¯_(ツ)_/¯

Backgrounds

Why **tnum**?

BPF Verifier

&

Safety

Division-by-0?

Address leakage?

Out-of-bound read?

BPF Verifier & Safety

Uninit stack?

Invalid return value?

Infinite loop?

Termination?

Is pointer aligned?

Out-of-bound write?

Division-by-0?

Out-of-bound read?

Uninit stack?

What's the **values** being **used?**

Invalid return value?

Infinite loop?

Termination?

Is pointer aligned?

Out-of-bound write?

```c
/* i is some random number */
int i = bpf_get_prandom_u32();
/* mask must be 3 */
int mask = 3;

/* i & mask can be 0, 1, 2, or 3 */
return i & mask;
```

What the verifier **actually** sees

```
/* random number given as the
 * return value (register r0) */
call bpf_get_prandom_u32;
/* mask stored in register r1*/
r1 = 3;

r0 &= r1; /* (i & mask) kept in r0 */
ret;
```

Division-by-0?

Out-of-bound read?

Uninit stack?

What's the **values** being **used?**

Invalid return value?

Infinite loop?

Termination?

Is pointer aligned?

Out-of-bound write?

Division-by-0?

Out-of-bound read?

Uninit stack?

What's the **value** within **registers?**

Invalid return value?

Infinite loop?

Termination?

Is pointer aligned?

Out-of-bound write?

Division-by-0?

Out-of-bound read?

Uninit stack?

Value Tracking

Invalid return value?

Infinite loop?

Termination?

Is pointer aligned?

Out-of-bound write?

Value Tracking

Attempt #1 - Naïve Approach

```
/* Takes 2^31 GiB just to track a
 * single register */
struct values {
 char possibly_0 :1;
 char possibly_1 :1;
 char possibly_2 :1;
 ...
}
```

Attempt #1 - Naïve Approach

```
/* Takes 2^31 GiB just to track a
 * single register */
struct values {
  char possibly_0 :1;
  char possibly_1 :1;
  char possibly_2 :1;
  ...
}
```

Attempt #1 - Naïve Approach

```
struct values add_values(struct values *a,
                                       struct values *b) {
    if (a->possibly_0 && b->possibly_0)
        ret->possibly_0 = 1;
    if (a->possibly_0 && b->possibly_1)
        ret->possibly_1 = 1;
    if (a->possibly_1 && b->possibly_0)
        ret->possibly_1 = 1;
    /* Some bit-tricks would help, but ... */
```

Attempt #1 - Naïve Approach

```
struct values add_values(struct values *a,
                         struct values *b) {
    if (a->possibly_0 && b->possibly_0)
      ret->possibly_0 = 1;
    if (a->possibly_0 && b->possibly_1)
      ret->possibly_1 = 1;
    if (a->possibly_1 && b->possibly_0)
      ret->possibly_1 = 1;
    /* Some bit-tricks would help, but ... */
```

Just track

min & **max**

Attempt #2 - Ranges

```
struct values {
  s64 min;
  s64 max;
};
```

Attempt #2 - Ranges

```
struct values add_values(struct values *a,
                         struct values *b)
{
  /* Ignoring overflow for now */
  ret->min = a->min + b->min;
  ret->max = a->max + b->max;
}
```

Attempt #2 - Ranges

```
struct bpf_reg_state {
  struct tnum var_off;
  s64 smin_value;
  s64 smax_value;
  ...
```

Attempt #2 - Ranges

```
struct bpf_reg_state {
  struct tnum var_off;
  s64 smin_value; /* minimum possible (s64)value */
  s64 smax_value; /* maximum possible (s64)value */
  u64 umin_value; /* minimum possible (u64)value */
  u64 umax_value; /* maximum possible (u64)value */
  s32 s32_min_value; /* minimum possible (s32)value */
  s32 s32_max_value; /* maximum possible (s32)value */
  u32 u32_min_value; /* minimum possible (u32)value */
  u32 u32_max_value; /* maximum possible (u32)value */
```

What about **bitwise** operations?

Attempt #2 - Ranges

```
struct values xor_values(struct values *a,
                         struct values *b)
{




}
```

Attempt #2 - Ranges

```
struct values xor_values(struct values *a,
                         struct values *b)
{

        ain't nobody got
        time for that¹

}
```

1: Okay, slightly mis-quoting here as the original actually refers to calculating signed-bounds in mul/and/or

Track individual **bits**

Attempt #3 - Bitwise Pattern

Each **bit** in the register can have **three** possible states:

- Unknown □ **`x`**
- Known to be set □ **`1`**
- Known to be unset □ **`0`**

Attempt #3 - Bitwise Pattern

Each **bit** in the register can have **three** possible states:

- Unknown □ **`x`** □ `{0, 1}`

- Known to be set □ **`1`**

- Known to be unset □ **`0`**

```
{ 1, 3 }
{ 0b01, 0b11 }
{ 0b01, 0b11 }
```

```
0t
```

```
{ 0b01, 0b11 }
    ↑     ↑
```

```
0tx
  ↑
```

```
{ 0b01, 0b11 }
```

```
0tx
```

```
{ 0b01, 0b11 }
```

```
0tx1
```

```
{ 0b01, 0b11 }
```

```
0tx1
```

Concrete

```
{ 0b01, 0b11 }
```

Abstract

```
0tx1
```

Concrete

(actual values used in register)

```
{ 0b01, 0b11 }
```

Abstract

(how we represent such set of values)

```
0tx1
```

Concrete

```
{ 0b0…01, 0b0…11 }
```

Abstract

```
0t0…x1
```

Concrete

```
{ 0b01, 0b11 }
```

Abstract

```
0tx1
```

Concrete

```
{ 1, 3 }
```

Abstract

```
0tx1
```

Concrete

```
{ 1, 3 }
```

non-consecutive values
(e.g. **pointer alignment**)

Abstract

```
0tx1
```

Limitation

Fuzzy

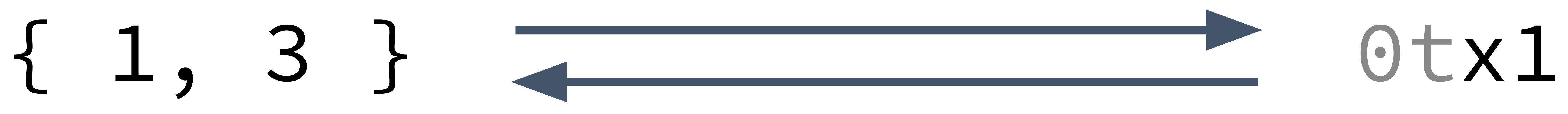
Concrete

`{ 1, 3 }`

Abstract

`0tx1`

Concrete | Abstract


$$\begin{tikzcd} \{\ 1,\ 3\ \} \arrow[r, shift left] & \texttt{0tx1} \arrow[l, shift left] \end{tikzcd}$$

Concrete

```
{ 1, 2 }
```

Abstract

Concrete

```
{ 0b01, 0b10 }
```

Abstract

Concrete | Abstract

$$\begin{tikzcd}\texttt{\{ 0b01, 0b10 \}} \arrow[r] & \texttt{0t}\end{tikzcd}$$

Concrete

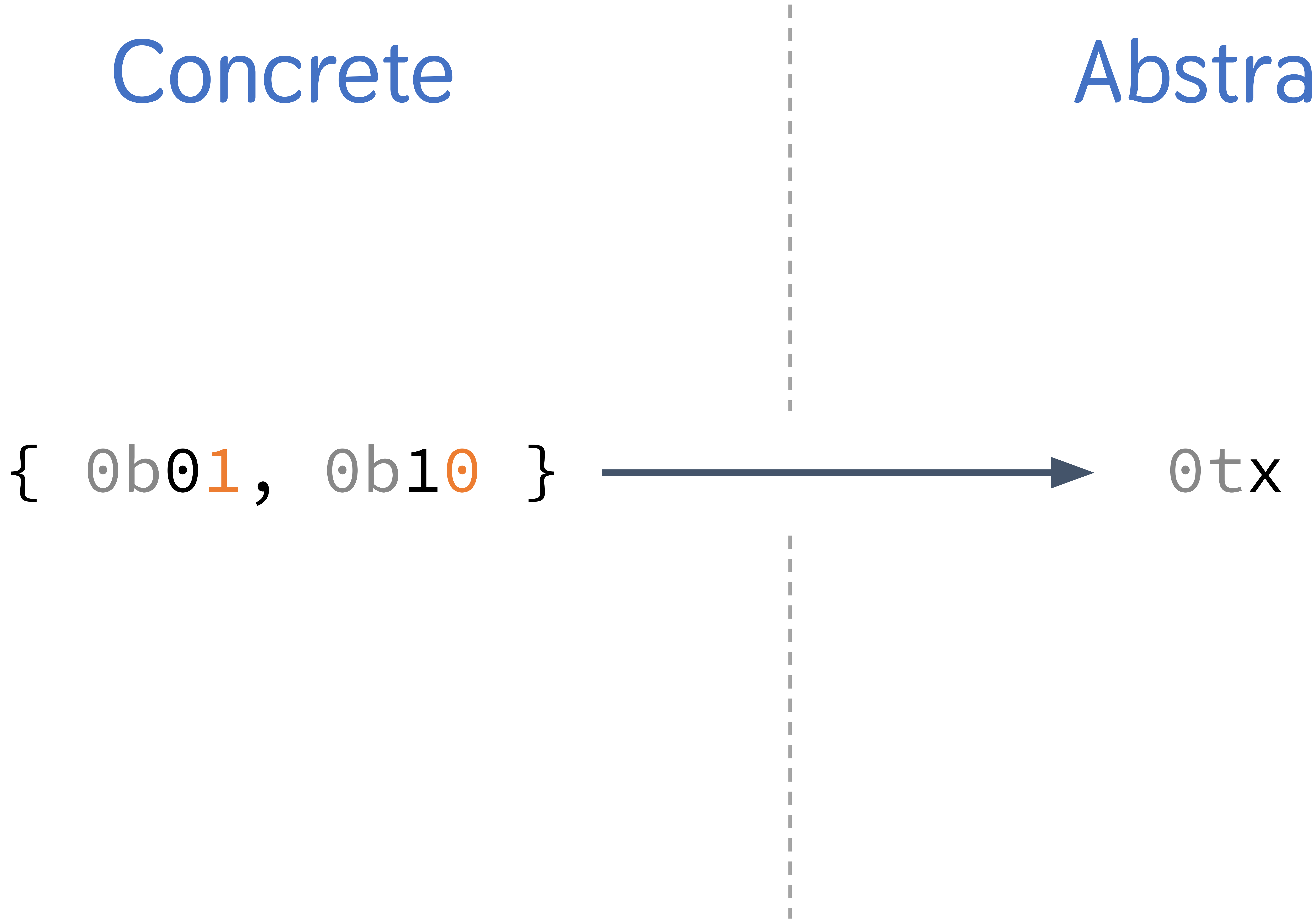
Abstract

$$\begin{tikzcd} \{\ \mathtt{0b01},\ \mathtt{0b10}\ \} \arrow[r] & \mathtt{0t} \end{tikzcd}$$

Concrete

Abstract

$$\texttt{\{ 0b01, 0b10 \}} \longrightarrow \texttt{0tx}$$


Concrete
Abstract
{ 0b01, 0b10 }
0tx

Concrete

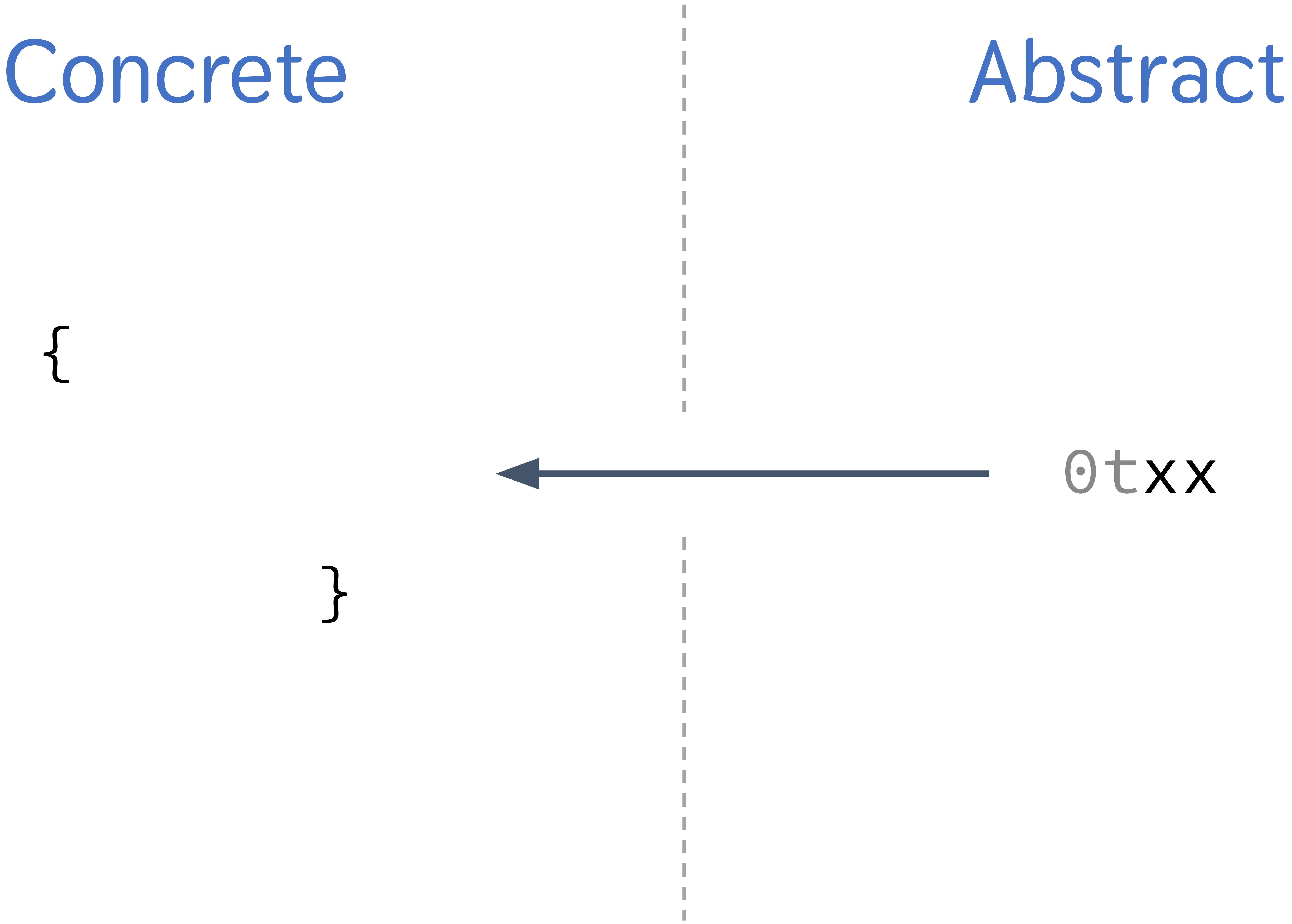
Abstract

$$\{\ \texttt{0b01},\ \texttt{0b10}\ \} \longrightarrow \texttt{0txx}$$

Concrete

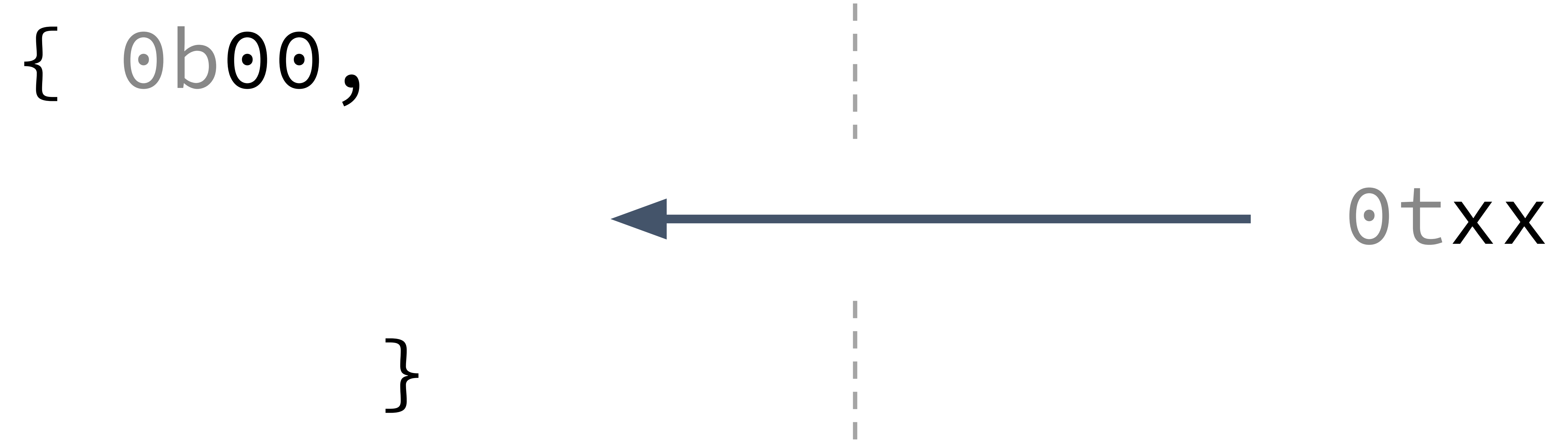
Abstract

$$\begin{tikzcd}
\{\ \texttt{0b01},\ \texttt{0b10}\ \} \arrow[r] & \texttt{0txx}
\end{tikzcd}$$


Concrete
Abstract
{
}
0txx

Concrete

Abstract



```
{ 0b00,
          }
```

```
0txx
```

Concrete | Abstract

```
{ 0b00,
  0b01,
                 <------------------  0txx

      }
```

Concrete

Abstract

```
{ 0b00,
  0b01,
  0b10,
       }
```

$\longleftarrow$

```
0txx
```

Concrete

Abstract

$$\begin{tikzcd} \{\texttt{0b00}, \texttt{0b01}, \texttt{0b10}, \texttt{0b11}\} & \texttt{0txx} \arrow[l] \end{tikzcd}$$

Concrete
(ideal)

Abstract

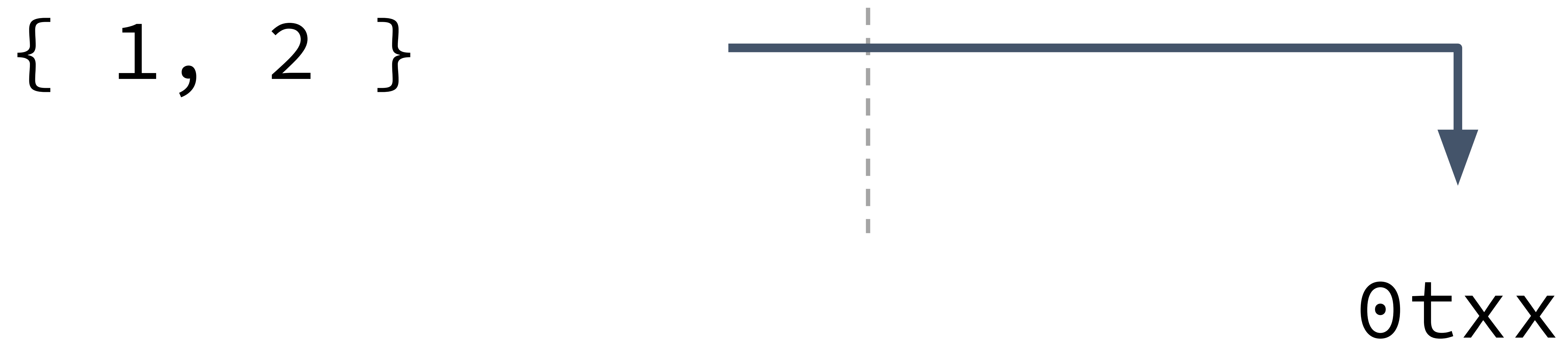
Concrete
(ideal)

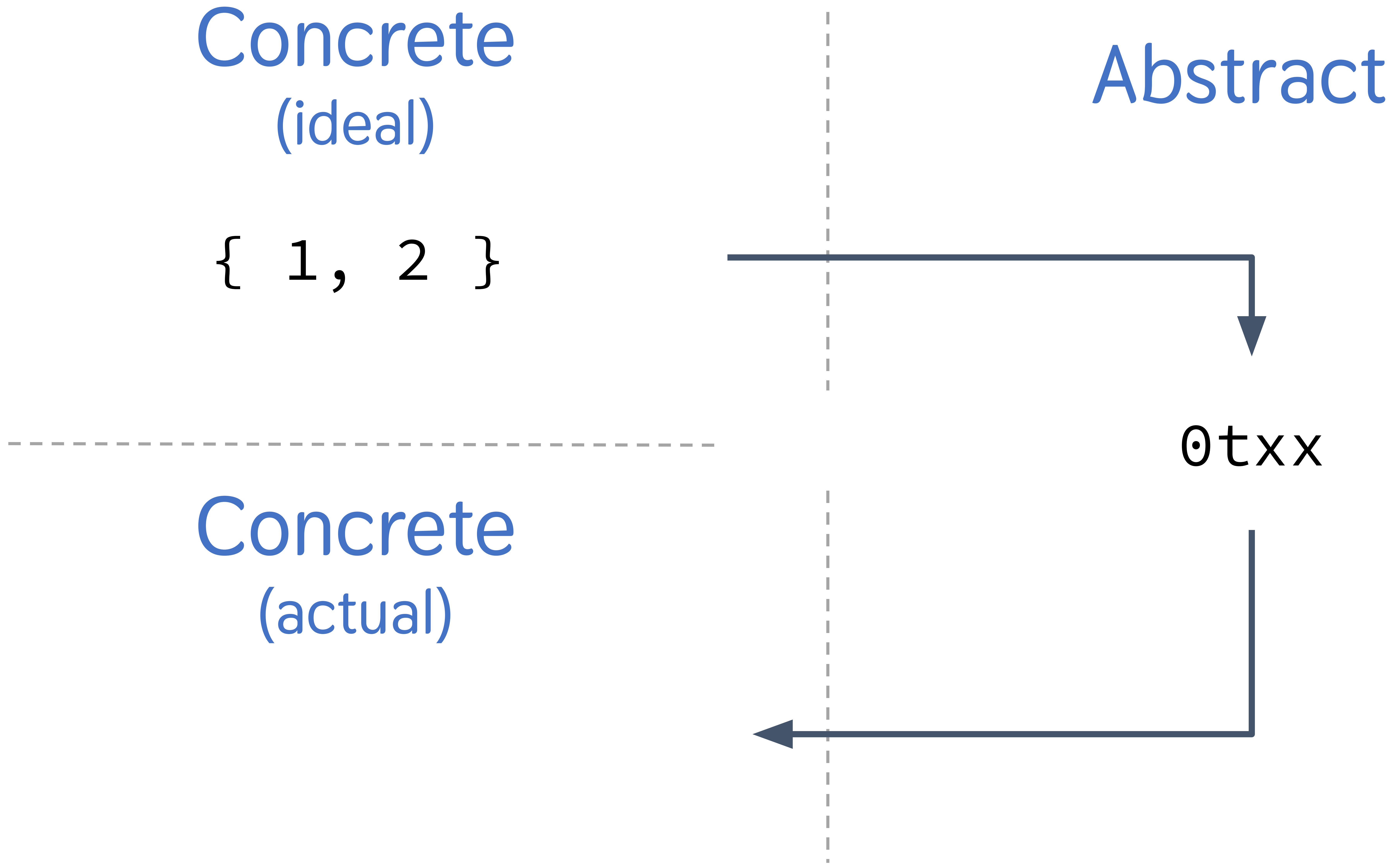
```
{ 1, 2 }
```

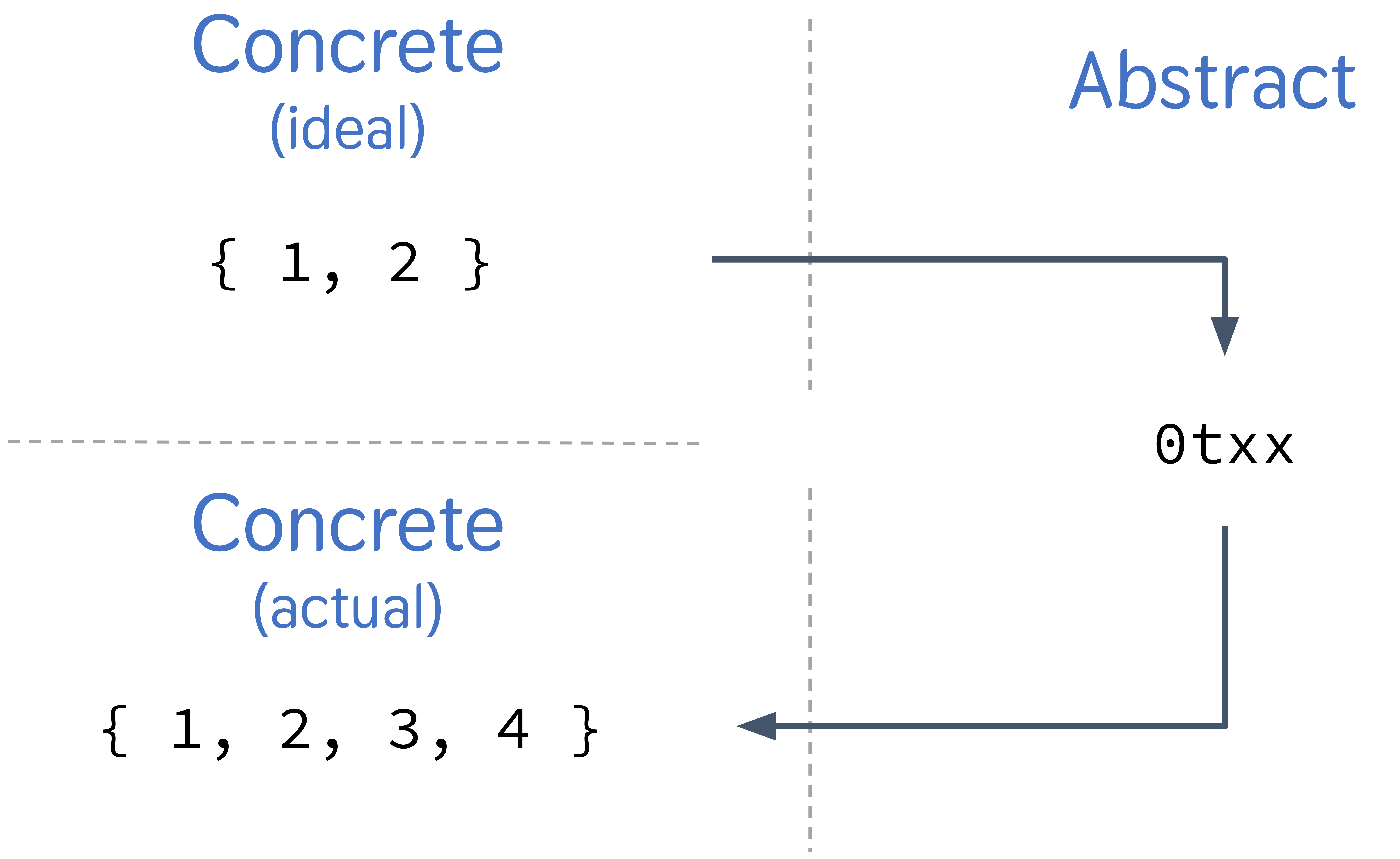
Abstract

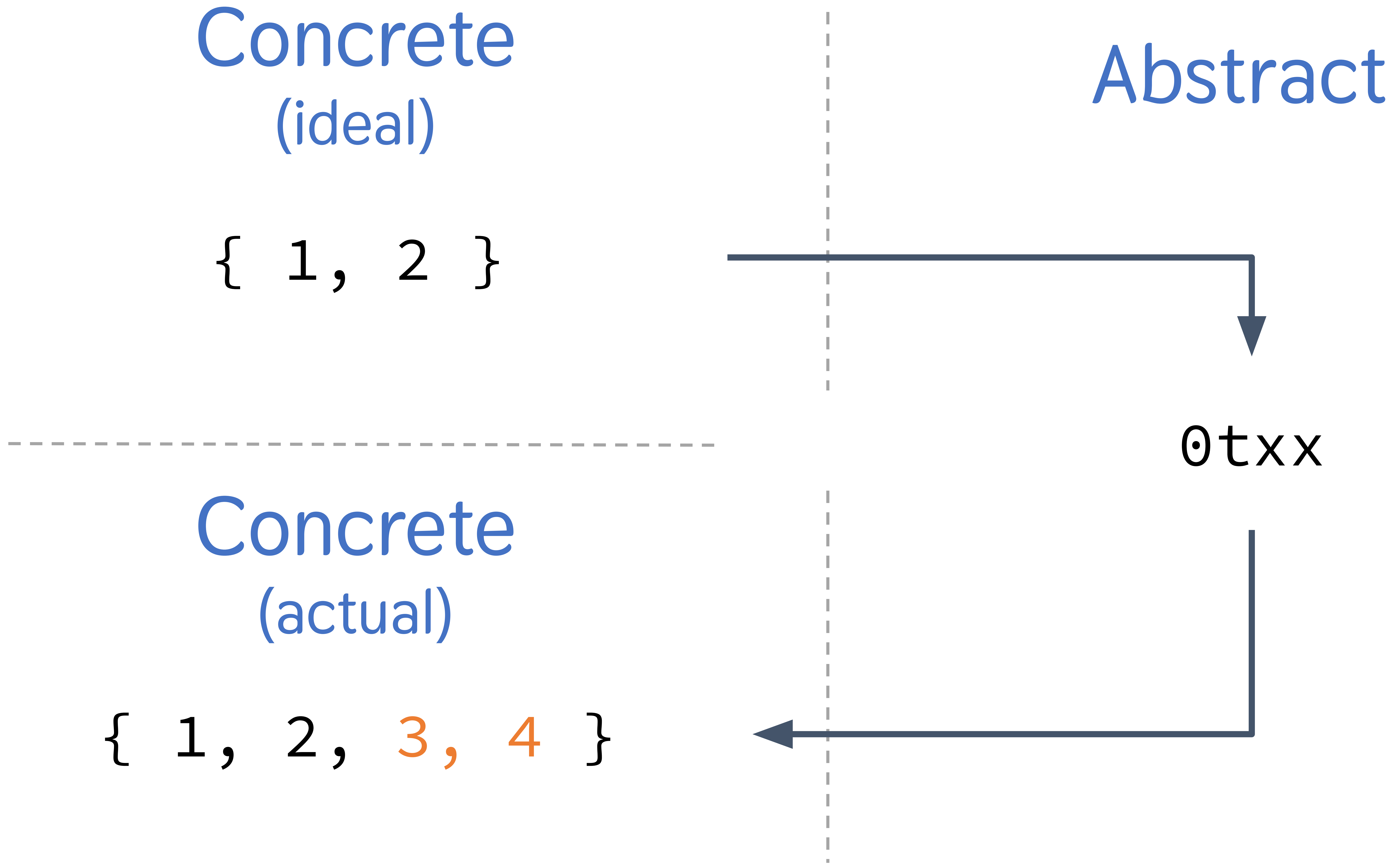
Concrete
(ideal)

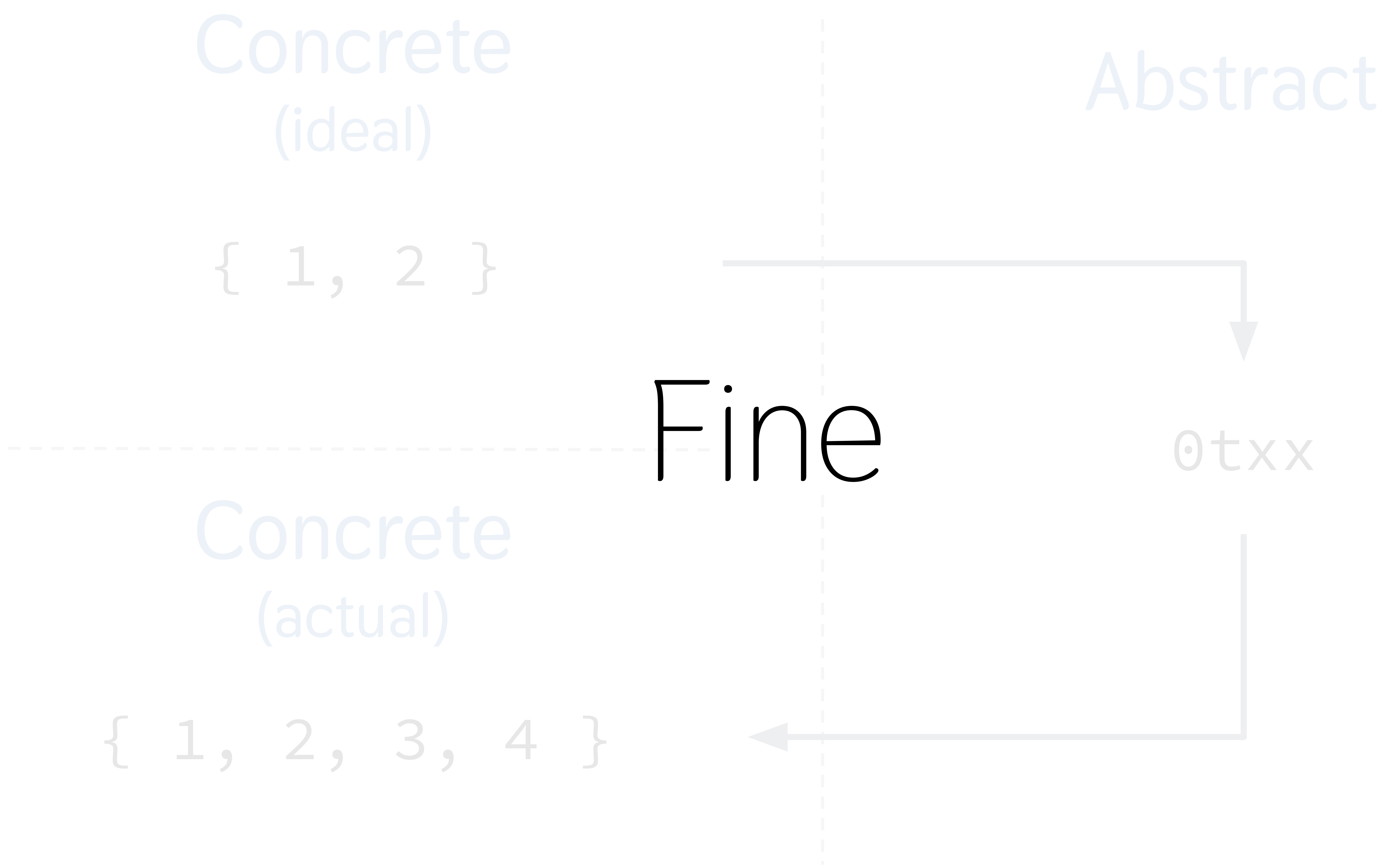
Abstract



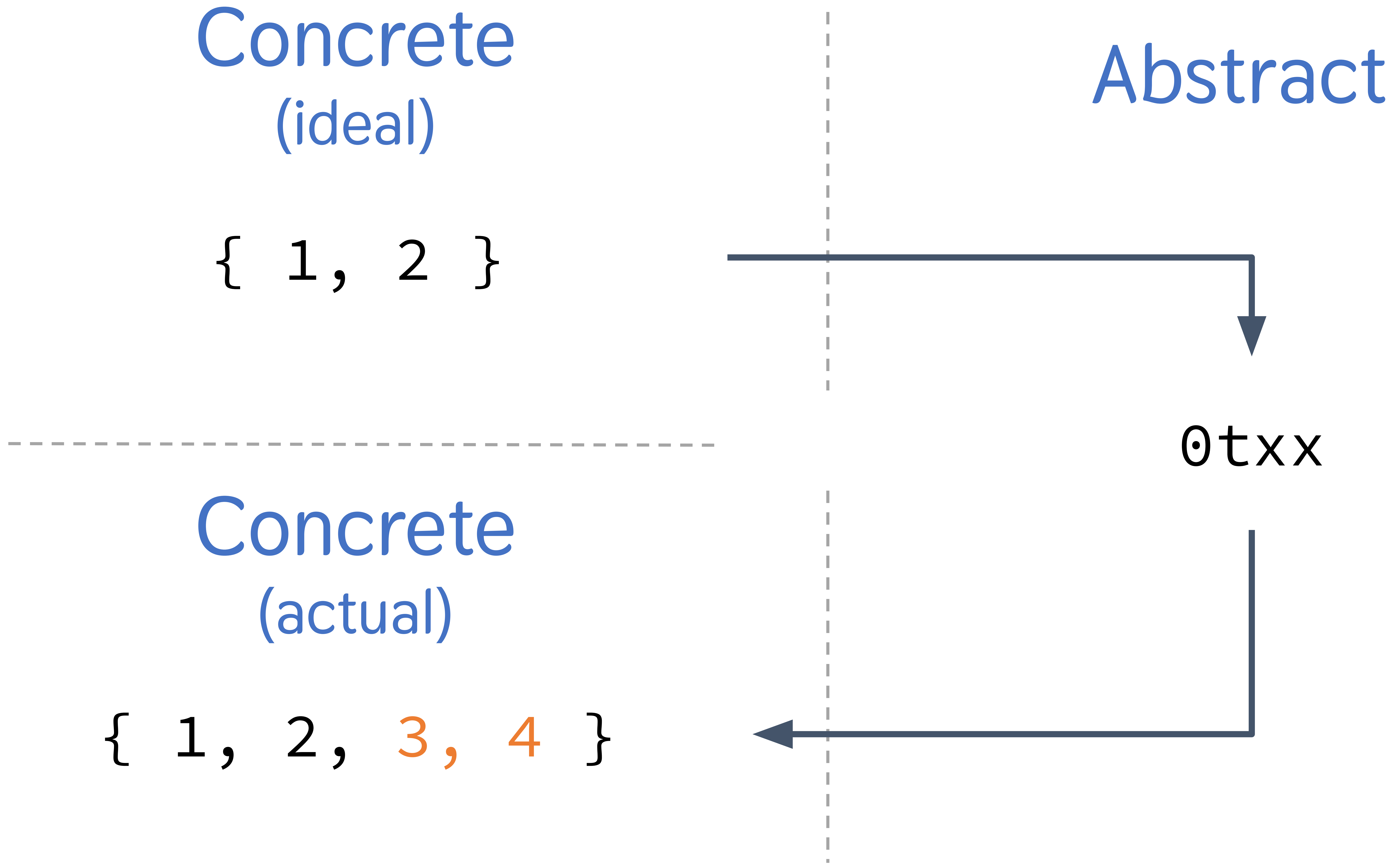

Concrete
(ideal)
{ 1, 2 }
Abstract
0txx
Concrete
(actual)

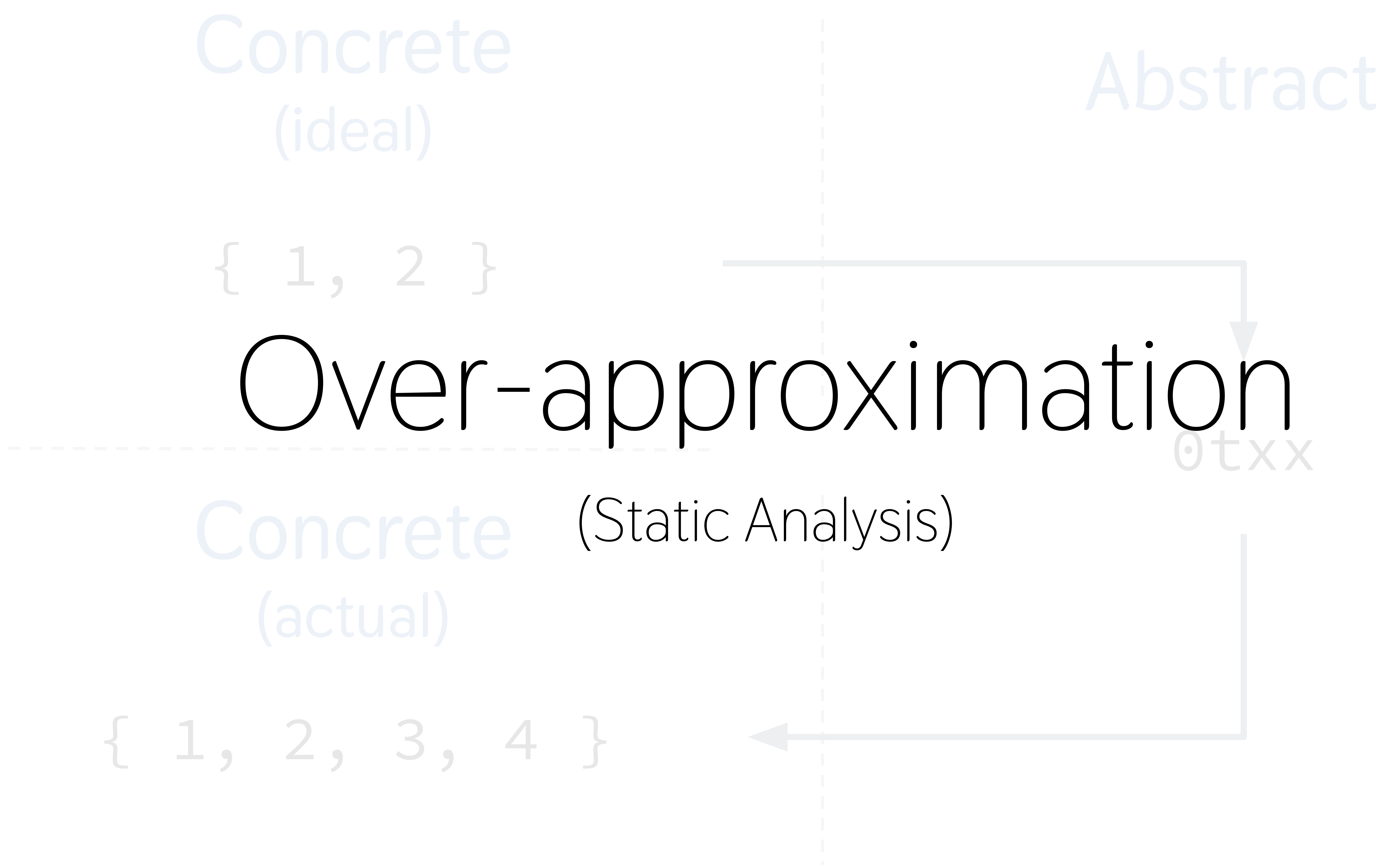

Concrete
(ideal)
{ 1, 2 }
Abstract
0txx
Concrete
(actual)
{ 1, 2, 3, 4 }


Concrete
(ideal)
{ 1, 2 }
Abstract
0txx
Concrete
(actual)
{ 1, 2, 3, 4 }



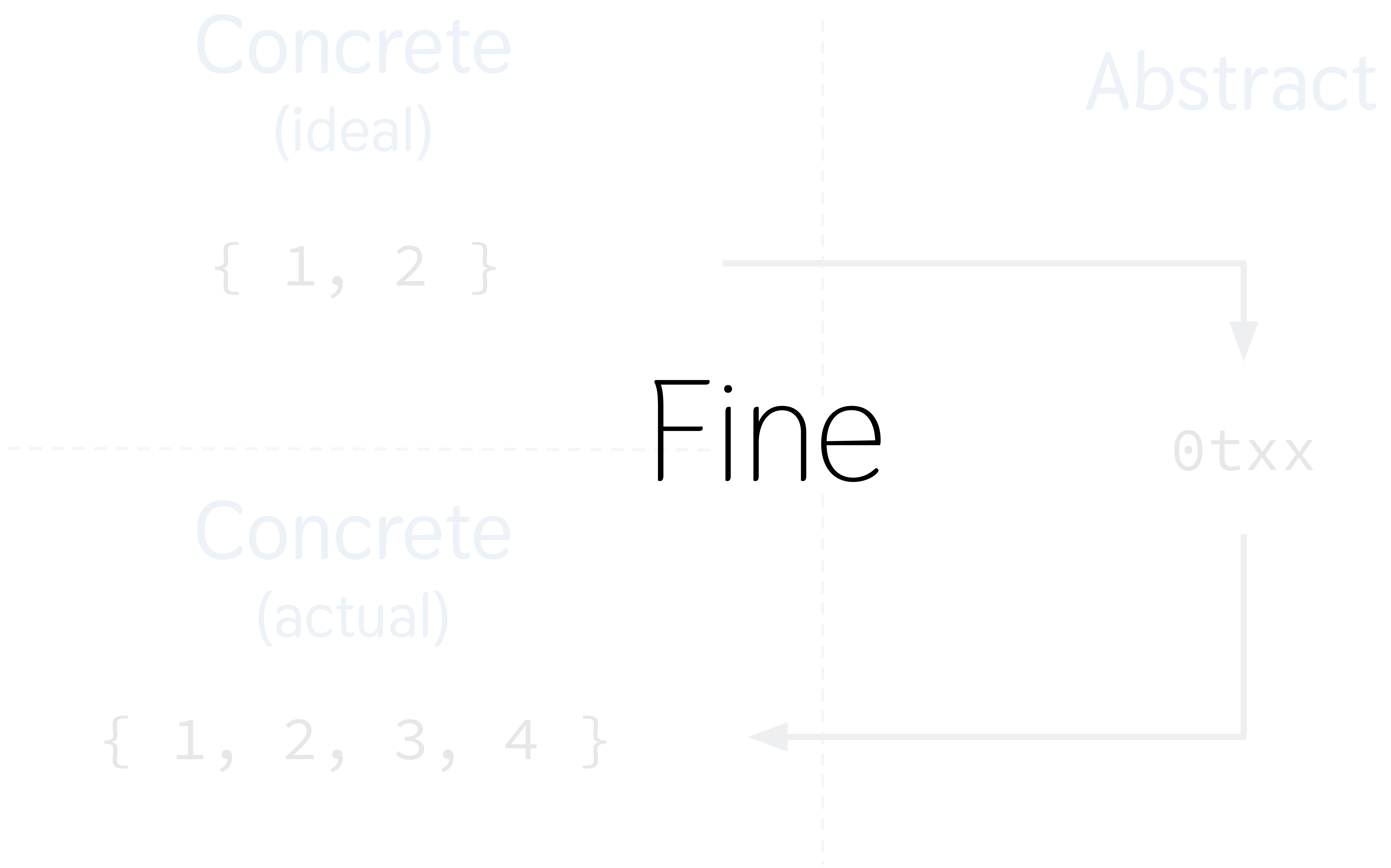
Fine


Concrete
(ideal)
{ 1, 2 }
Abstract
0txx
Concrete
(actual)
{ 1, 2, 3, 4 }



Over-approximation

(Static Analysis)



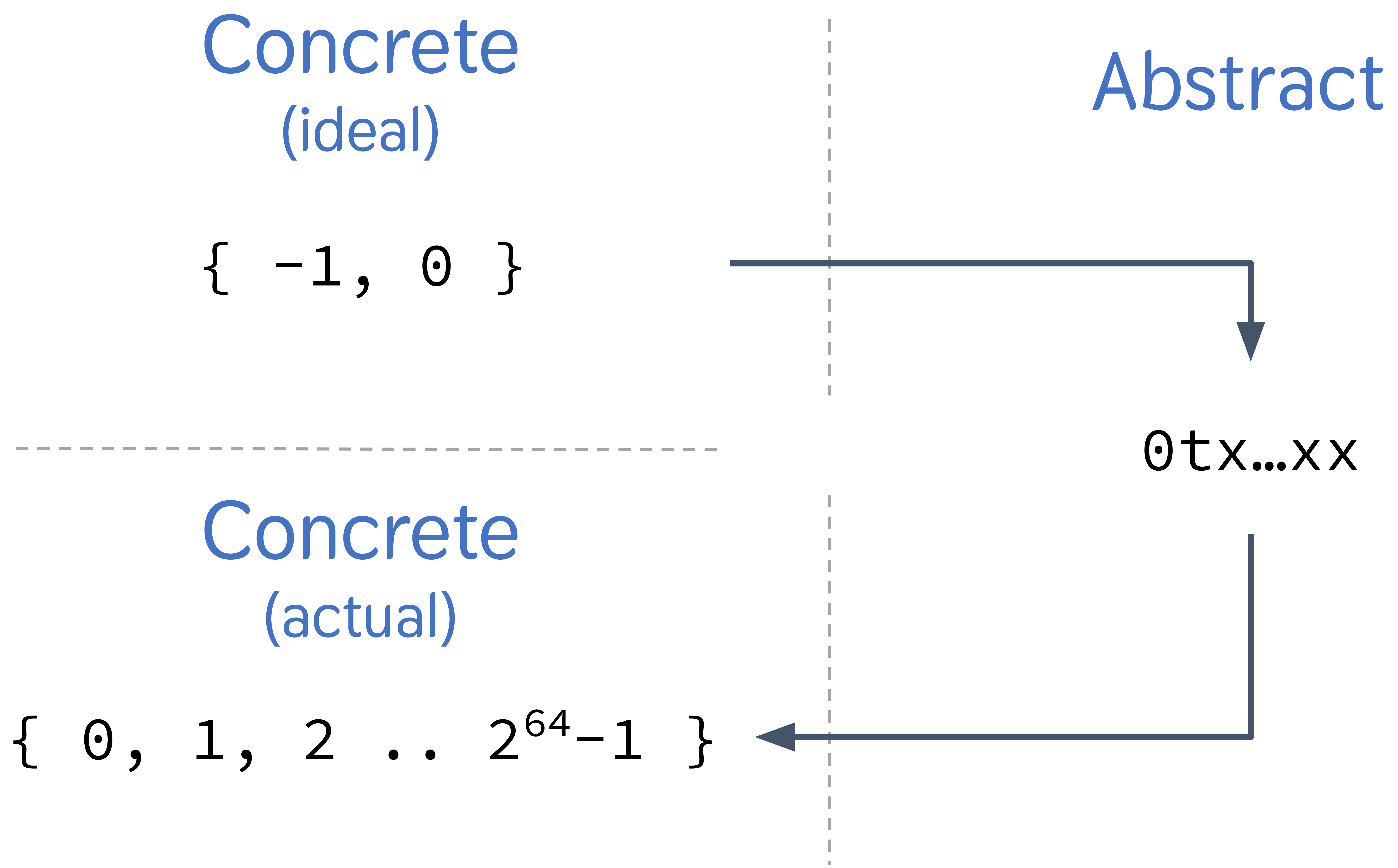
Fine

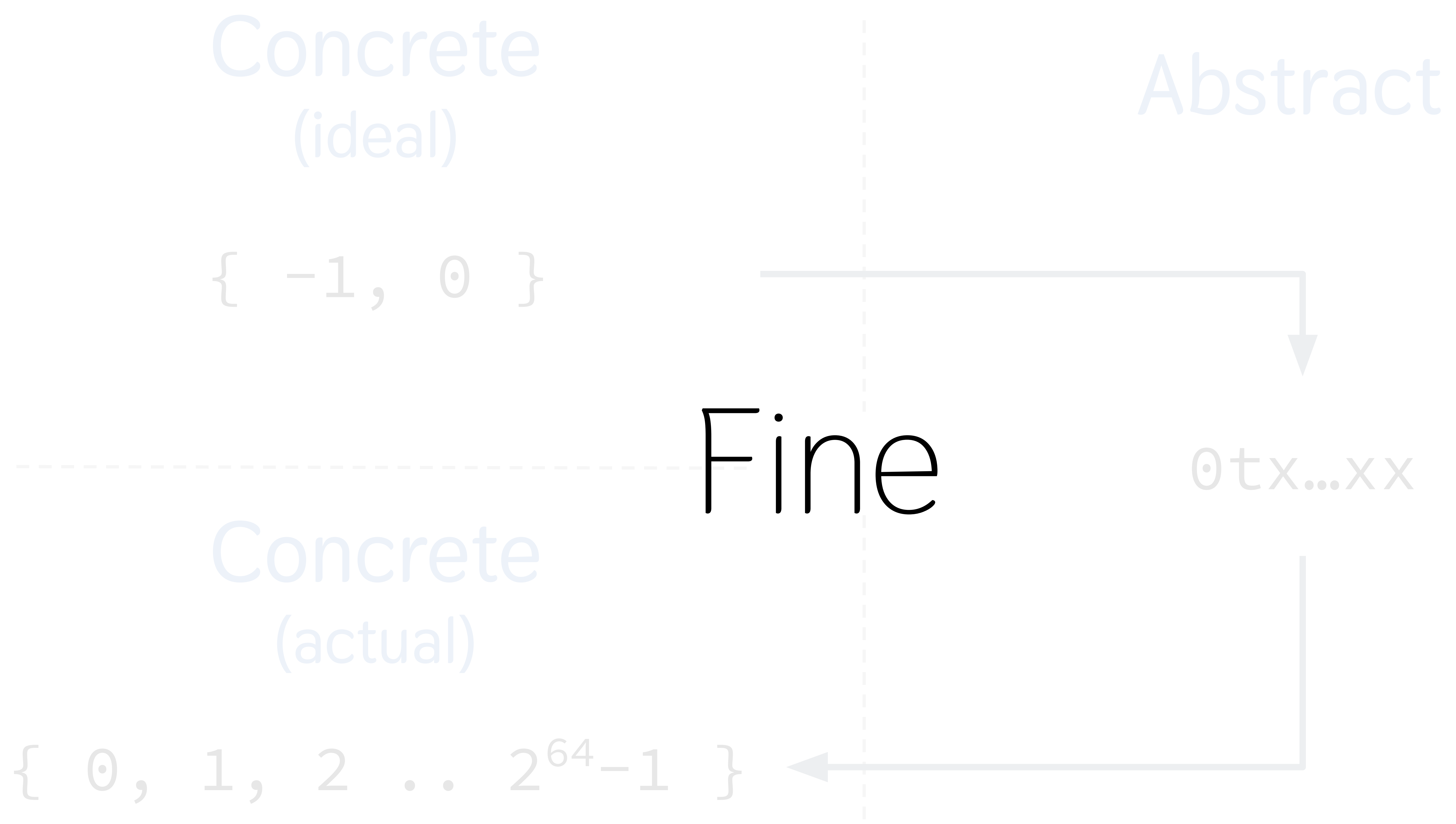
Attempt #2 - Ranges

```
struct bpf_reg_state {
  struct tnum var_off;
  s64 smin_value; /* minimum possible (s64)value */
  s64 smax_value; /* maximum possible (s64)value */
  u64 umin_value; /* minimum possible (u64)value */
  u64 umax_value; /* maximum possible (u64)value */
  s32 s32_min_value; /* minimum possible (s32)value */
  s32 s32_max_value; /* maximum possible (s32)value */
  u32 u32_min_value; /* minimum possible (u32)value */
  u32 u32_max_value; /* maximum possible (u32)value */
```

Signess

crossing sign boundary


Concrete
(ideal)
Abstract
{ -1, 0 }
0tx…xx
Concrete
(actual)
{ 0, 1, 2 .. 2^64-1 }



Fine

Attempt #2 - Ranges

```
struct bpf_reg_state {
  struct tnum var_off;
  s64 smin_value; /* minimum possible (s64)value */
  s64 smax_value; /* maximum possible (s64)value */
  u64 umin_value; /* minimum possible (u64)value */
  u64 umax_value; /* maximum possible (u64)value */
  s32 s32_min_value; /* minimum possible (s32)value */
  s32 s32_max_value; /* maximum possible (s32)value */
  u32 u32_min_value; /* minimum possible (u32)value */
  u32 u32_max_value; /* maximum possible (u32)value */
```

Can't track **nothing**[1]

$\varnothing$

1: We could make a currently unused and invalid representation of tnum (e.g. `val = -1 && mask = -1`) to mean an empty set, but might *not* be a good idea.

```c
/* assume this isn’t optimized out */
if (i < 0 && i > 0) {
  /* never ever */
}
```

```
/* assume this isn't optimized out */
if (i < 0 && i > 0) {
  /* IMPOSSIBLE(*) to represent i */
}
```

Just **don’t follow** such **branch**

```
/* compute branch direction of the expression "if
 * (<reg1> opcode <reg2>) goto target;" and return:
 *  1 - branch will be taken
 *  0 - branch will not be taken
 * -1 - unknown. Example: "if (reg1 < 5)" is unknown
 *      when register value range [0,10]
 */
static int is_branch_taken(struct bpf_reg_state *reg1,
                           struct bpf_reg_state *reg2,
                           u8 opcode, bool is_jmp32);
```

```
static int is_scalar_branch_taken(...) {
 switch (opcode) {
 case BPF_JEQ:
  if (tnum_is_const(t1) && tnum_is_const(t2))
   return t1.value == t2.value;
  ...
  return -1;
 ...
```

Can’t track **relation**

```
int j = i - 1; /* int i is unknown */

if (i < 1 || i > 3)
  return;
/* From here on 1 ≤ i ≤ 3
/* with j == i - 1 we know 0 ≤ j ≤ 2
 */
if (j == 4)
  /* never ever */
```

Track **relationship** separately

```
struct bpf_reg_state {
  /* Upper bit of ID is used to remember relationship
   * between "linked" registers, e.g.:
   * r1 =  r2;   both will have r1->id == r2->id == N
   * r1 += 10;   r1->id == N | BPF_ADD_CONST and
   *             r1->off == 10
   */
#define BPF_ADD_CONST (1U << 31)
  u32 id;
  ...
```

Implementation

Data Structure

Concrete

```
{ 0b01, 0b11 }
```

Abstract

```
0tx1
```

Concrete

Abstract

Conceptual

Implementation

```
{ 0b01, 0b11 }
```

```
0tx1
```

```
struct tnum {
 u64 value; /* whether bits are
             * set/unset, if known
             */
 u64 mask;  /* which bits are
            * unknown */
};
```

Each **bit** in the register can have **three** possible states:

- Unknown □ **`x`**
- Known to be set □ **`1`**
- Known to be unset □ **`0`**

Each **bit** in the register can have **three** possible states:

- Unknown □ **`x`**

- Known to be set □ **`1`** `(mask[] = 0, value[] = 1)`

- Known to be unset □ **`0`**

Each **bit** in the register can have **three** possible states:

- Unknown □ **`x`**

- Known to be set □ **`1`**

- Known to be unset □ **`0`** `(mask[] = 0,`
  `value[] = 0)`

Each **bit** in the register can have **three** possible states:

- Unknown □ **`x`** `(mask[] = 1)`
- Known to be set □ **`1`**
- Known to be unset □ **`0`**

Each **bit** in the register can have **three** possible states:

- Unknown □ **x** (`mask[] = 1, value[] = 0`)
- Known to be set □ **`1`**
- Known to be unset □ **`0`**

Each **bit** in the register can have **three** possible states:

- Unknown □ **`x`**
- Known to be set □ **`1`**
- Known to be unset □ **`0`**
- Invalid □ `(mask□ = 1, value□ = 1)`

Concrete

```
{ 0b01, 0b11 }
```

Abstract

Conceptual

```
0tx1
```

Implementation

```
.mask  = 0b
.value = 0b
```

LINUX PLUMBERS CONFERENCE Vienna, Austria / Sept. 18-20, 2024

Concrete

```
{ 0b01, 0b11 }
```

Abstract

Conceptual

```
0tx1
```

Implementation

```
.mask  = 0b
.value = 0b
```

Concrete

```
{ 0b01, 0b11 }
```

Abstract

Conceptual

```
0tx1
```

Implementation

```
.mask  = 0b1
.value = 0b
```

Concrete

```
{ 0b01, 0b11 }
```

Abstract

Conceptual

```
0tx1
```

Implementation

```
.mask  = 0b1
.value = 0b0
```

Concrete

```
{ 0b01, 0b11 }
```

Abstract

Conceptual

```
0tx1
```

↑

Implementation

```
.mask  = 0b1
.value = 0b0
```

Concrete

```
{ 0b01, 0b11 }
```

Abstract

Conceptual

```
0tx1
```

Implementation

```
.mask  = 0b10
.value = 0b0
```

Concrete

```
{ 0b01, 0b11 }
```

Abstract

Conceptual

```
0tx1
```

Implementation

```
.mask  = 0b10
.value = 0b01
```

Concrete

```
{ 0b01, 0b11 }
```

Abstract

Conceptual

```
0tx1
```

Implementation

```
.mask  = 0b10
.value = 0b01
```

Concrete

```
{ 1, 3 }
```

Abstract

Conceptual

```
0tx1
```

Implementation

```
.mask  = 0b10
.value = 0b01
```

Implementation

Helper

```
u64 tnum_umin(struct tnum a)
```

minimum possible unsigned value in a tnum

```
a.value
```

```
u64 tnum_umax(struct tnum a)
```

maximum possible unsigned value in a tnum

```
a.value | a.mask
```

```
u64 tnum_and(struct tnum a,
             struct tnum b)
```

bitwise-and of two tnums

Crafting

How **well** do you need to **know tnum**?

to craft an operator

Very little

```
u64 tnum_and(struct tnum a,
             struct tnum b)
```

bitwise-and of two tnums

a

	&	0	1
b	0		
	1		

&	0	1
0	0 & 0	0 & 1
1	1 & 0	1 & 1

&	**0**	**1**
0	0	0
1	0	1

&	0	1	x
0	0	0	
1	0	1	
x			

&	0	1	x
0	0	0	
1	0	1	
x	?		

&	0	1	x
0	0	0	
1	0	1	
x			

&	0	1	x
0	0	0	
1	0	1	
x	0		

&	0	1	x
0	0	0	
1	0	1	
x	0		

&	0	1	x
0	0	0	
1	0	1	
x	0	{0, 1}	

&	0	1	x
0	0	0	
1	0	1	
x	0	x	

&	0	1	x
0	0	0	0
1	0	1	
x	0	x	

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	?

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	{0, 1}

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

&	m=0 v=0	1	x
m=0 v=0	m=0 v=0	m=0 v=0	m=0 v=0
1	m=0 v=0	1	x
x	m=0 v=0	x	x

&	m=0 v=0	1	x
m=0 v=0	m=0 v=0	m=0 v=0	m=0 v=0
1	m=0 v=0	1	x
x	m=0 v=0	x	x

&	**m=0 v=0**	**m=0 v=1**	**x**
m=0 v=0	m=0 v=0	m=0 v=0	m=0 v=0
m=0 v=1	m=0 v=0	m=0 v=1	x
x	m=0 v=0	x	x

&	m=0
	v=0
		m=0
		v=1
			x

m=0
v=0
	m=0
	v=0
		m=0
		v=0
			m=0
			v=0

m=0
v=1
	m=0
	v=0
		m=0
		v=1
			x

x	m=0
	v=0
		x	x

&	m=0
	v=0
		m=0
		v=1
			m=1
			v=0

m=0
v=0
	m=0
	v=0
		m=0
		v=0
			m=0
			v=0

m=0
v=1
	m=0
	v=0
		m=0
		v=1
			m=1
			v=0

m=1
v=0
	m=0
	v=0
		m=1
		v=0
			m=1
			v=0

&	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	v=0	v=0	v=0
m=0 v=1	v=0	v=1	v=0
m=1 v=0	v=0	v=0	v=0

&	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	m=0	m=0	m=0
m=0 v=1	m=0	m=0	m=1
m=1 v=0	m=0	m=1	m=1

&.v	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	1	0
m=1 v=0	0	0	0

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

&.v	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	1	0
m=1 v=0	0	0	0

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

&.v	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	1	0
m=1 v=0	0	0	0

&.v	m=0
	v=0
		m=0
		v=1
			m=1
			v=0

m=0
v=0
	0	0	0

m=0
v=1
	0	1	0

m=1
v=0
	0	0	0

&.v	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	1	0
m=1 v=0	0	0	0

```
value =
 a.value & b.value
```

&.v	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	1	0
m=1 v=0	0	0	0

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

```
mask =
 (a.value | a.mask)
         &
 (b.value | b.mask)
```

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

&.m	m=0
	v=0
		m=0
		v=1
			m=1
			v=0

m=0
v=0
	0	0	0

m=0
v=1
	0	0	1

m=1
v=0
	0	1	1

a.value & b.value

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

```
mask =
 (a.value | a.mask)
          &
 (b.value | b.mask)
          &
 ~(a.value & b.value)
```

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

```
struct tnum tnum_and(struct tnum a, struct tnum b)
{
 u64 alpha, beta, v;

 alpha = a.value | a.mask;
 beta = b.value | b.mask;
 v = a.value & b.value;
 return TNUM(v, alpha & beta & ~v);
}
```

```c
struct tnum tnum_and(struct tnum a, struct tnum b)
{
 u64 alpha, beta, v;

 alpha = a.value | a.mask;
 beta = b.value | b.mask;
 v = a.value & b.value;
 return TNUM(v, alpha & beta & ~v);
}
```

&.v	m=0
	v=0
		m=0
		v=1
			m=1
			v=0

m=0
v=0
	0	0	0

m=0
v=1
	0	1	0

m=1
v=0
	0	0	0

```
value =
  a.value & b.value
```

```c
struct tnum tnum_and(struct tnum a, struct tnum b)
{
 u64 alpha, beta, v;

 alpha = a.value | a.mask;
 beta = b.value | b.mask;
 v = a.value & b.value;
 return TNUM(v, alpha & beta & ~v);
}
```

```
struct tnum tnum_and(struct tnum a, struct tnum b)
{
 u64 alpha, beta, v;

 alpha = a.value | a.mask;
 beta = b.value | b.mask;
 v = a.value & b.value;
 return TNUM(v, alpha & beta & ~v);
}
```

```
mask =
 (a.value | a.mask)
           &
 (b.value | b.mask)
           &
 ~(a.value & b.value)
```

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

```c
struct tnum tnum_and(struct tnum a, struct tnum b)
{
 u64 alpha, beta, v;

 alpha = a.value | a.mask;
 beta = b.value | b.mask;
 v = a.value & b.value;
 return TNUM(v, alpha & beta & ~v);
}
```

```
mask =
 (a.value | a.mask)
            &
 (b.value | b.mask)
            &
 ~(a.value & b.value)
```

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

```c
struct tnum tnum_and(struct tnum a, struct tnum b)
{
 u64 alpha, beta, v;

 alpha = a.value | a.mask;
 beta = b.value | b.mask;
 v = a.value & b.value;
 return TNUM(v, alpha & beta & ~v);
}
```

```
mask =
 (a.value | a.mask)
            &
 (b.value | b.mask)
            &
 ~(a.value & b.value)
```

&.m	m=0 v=0	m=0 v=1	m=1 v=0
m=0 v=0	0	0	0
m=0 v=1	0	0	1
m=1 v=0	0	1	1

```
struct tnum tnum_and(struct tnum a, struct tnum b)
{
 u64 alpha, beta, v;

 alpha = a.value | a.mask;
 beta = b.value | b.mask;
 v = a.value & b.value;
 return TNUM(v, alpha & beta & ~v);
}
```

```
u64 tnum_and(struct tnum a,
             struct tnum b)
```

bitwise-and of two tnums

```c
/* Return @a with lowest @size bytes
 * retained, and all other bits set
 * to equal the sign bit (which might
 * be unknown).
 */
struct tnum tnum_scast(struct tnum a,
                       u8 size)
```

Usage

Bound-syncing

```c
static void reg_bounds_sync(struct bpf_reg_state *reg)
{
  /* tnum -> u64, s64, u32, s32 */
  __update_reg_bounds(reg);
  /* u64 -> u32, s32; s64 -> u32, s32
   * u64 -> s64; s64 -> u64
   * u32 -> u64, s64; s32 -> u64, s64 */
  __reg_deduce_bounds(reg);
  __reg_deduce_bounds(reg); /* 2nd time */
  /* u64 -> tnum; u32 -> tnum */
  __reg_bound_offset(reg);
  /* tnum -> u64, s64, u32, s32 */
  __update_reg_bounds(reg);
}
```

```
static void __update_reg64_bounds(struct bpf_reg_state *reg)
{
  /* min signed is max(sign bit) | min(other bits) */
  reg->smin_value = max_t(s64, reg->smin_value,
        reg->var_off.value | (reg->var_off.mask & S64_MIN));
  /* max signed is min(sign bit) | max(other bits) */
  reg->smax_value = min_t(s64, reg->smax_value,
        reg->var_off.value | (reg->var_off.mask & S64_MAX));
  reg->umin_value = max(reg->umin_value, reg->var_off.value);
  reg->umax_value = min(reg->umax_value,
             reg->var_off.value | reg->var_off.mask);
}
```

Testing

Does it **work?**

Is it **correct?**

Would it allow

unsafe program

to pass?

BPF selftests

Agni

Z3Py

Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers

Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte

Concrete

Abstract

$$\begin{tikzcd}
\{1, 3\} + \{3, 5\} \arrow[d] & 0t0x1 + 0txx1 \arrow[d] \\
\{4, 6, 8\} & 0txx0 \arrow[dl] \\
\{2, 4, 6, 8\} &
\end{tikzcd}$$

Concrete | Abstract

$$\begin{tikzcd}
\{\ 1,\ 3\ \} + \{\ 3,\ 5\ \} \arrow[d] & \texttt{0t0x1} + \texttt{0txx1} \arrow[d] \\
\{\ 4,\ 6,\ 8\ \} & \texttt{0txx0} \arrow[dl] \\
\{\ 2,\ 4,\ 6,\ 8\ \} &
\end{tikzcd}$$

Would it allow

unsafe program

to pass?

```
struct tnum dont_know(struct tnum a,
                      struct tnum b)
{
  /* Jon Snow knows nothing */
  return tnum_unknown;
}
```

Would it reject
safe program
(too often)?

BPF selftests

Agni

Conclusion

Tracks bit pattern

- Simple (maybe not intuitively easy to understand)
- **Can't track** min/max/sign-crossing **precisely**

Correct operation should

- **Not left any possible values** out (i.e. sound)
- Tries to exclude as much impossible values (i.e. precise)
  - without introducing unnecessary complexity

Resources

- **Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers**
- Peeking into the BPF verifier
- More than you want to know about BPF verifier
- Value Tracking in BPF verifier
- Model Checking (a very small part) of BPF Verifer