SMT solvers and quantifiers

Chris Hawblitzel (and the Verus Team)

Microsoft Research

https://verus.rs/

Verification condition generation

test.rs → Verus → root.smt2 → SMT solver (Z3)

SMT file contains
verification conditions for test.rs

```
proof fn test(x: int, y: int)
    ensures x + y == y + x
{
}
```

verus smt1.rs --log-all

```
;; note: hand edited to simplify

(declare-const x! Int)
(declare-const y! Int)
(assert
 (not
   (= (Add x! y!) (Add y! x!))
))
(check-sat)
```

Try to find counterexample
(x and y such that “ensures”
does **not** hold)

Decidable and undecidable theories

DECIDABLE THEORIES

boolean expressions e ::= true | false | !e | e && e | e ==> e | ...

linear integer arithmetic e ::= ... | -2 | -1 | 0 | 1 | 2 | e + e | e – e | e <= e | ...

bit vector arithmetic e ::= ... | e & e | e << e | ...

uninterpreted functions e ::= ... | f(e,...,e)

datatypes e ::= ... | S(e, ..., e) | S.f | match e { ... }

UNDECIDABLE

nonlinear integer arithmetic e ::= ... | e * e | e / e | e % e

quantifiers e ::= ... | (forall|x: t| e) | (exists|x: t| e)

What algorithms solve the decidable theories?
What heuristics can help with the undecidable components?

SMT components used by Verus

Core Components (always enabled)

SAT solver

true | false | !e
e && e | …

Linear int arith

-2 | -1 | 0 | 1 | 2
| e + e | e – e
| e <= e | …

Uninterp funcs

spec fn f(x,…,x);
f(e,…,e)

Datatypes

struct S { … }
enum E { … }
S(e,…,e) | e.f | …

Quantifiers

forall|x: t| e
exists|x: t| e

- undecidable, but really important
- programmable by triggers

Additional Components (via "assert by")

Bit vector arith

e & e| e << e |
…

decidable, but can be slow

Nonlin int arith

e * e
e / e | e % e

undecidable,
often unpredictable

Congruence for uninterpreted functions

Uninterp funcs

spec fn f(x,…,x);
f(e,…,e)

Congruence principle:
equal inputs imply equal output

$$\left.\begin{array}{l} e1 == e1' \\ e2 == e2' \\ \ldots \\ en == en' \end{array}\right\} \text{==>}\ f(e1,\ldots,en) == f(e1',\ldots,en')$$

Example algorithm: linear arithmetic

Linear int arith

-2 | -1 | 0 | 1 | 2
| e + e | e – e
| e <= e | …

Example: (4*x + 2*y - 2*z <= 0) && (3*x + 2*y - z <= 0)
&& (-4*x - 3*y + z <= 0) && (-5*x - 4*y + z <= 0)

Sample algorithm: "Fourier-Motzkin elimination"

Eliminate one variable at a time:

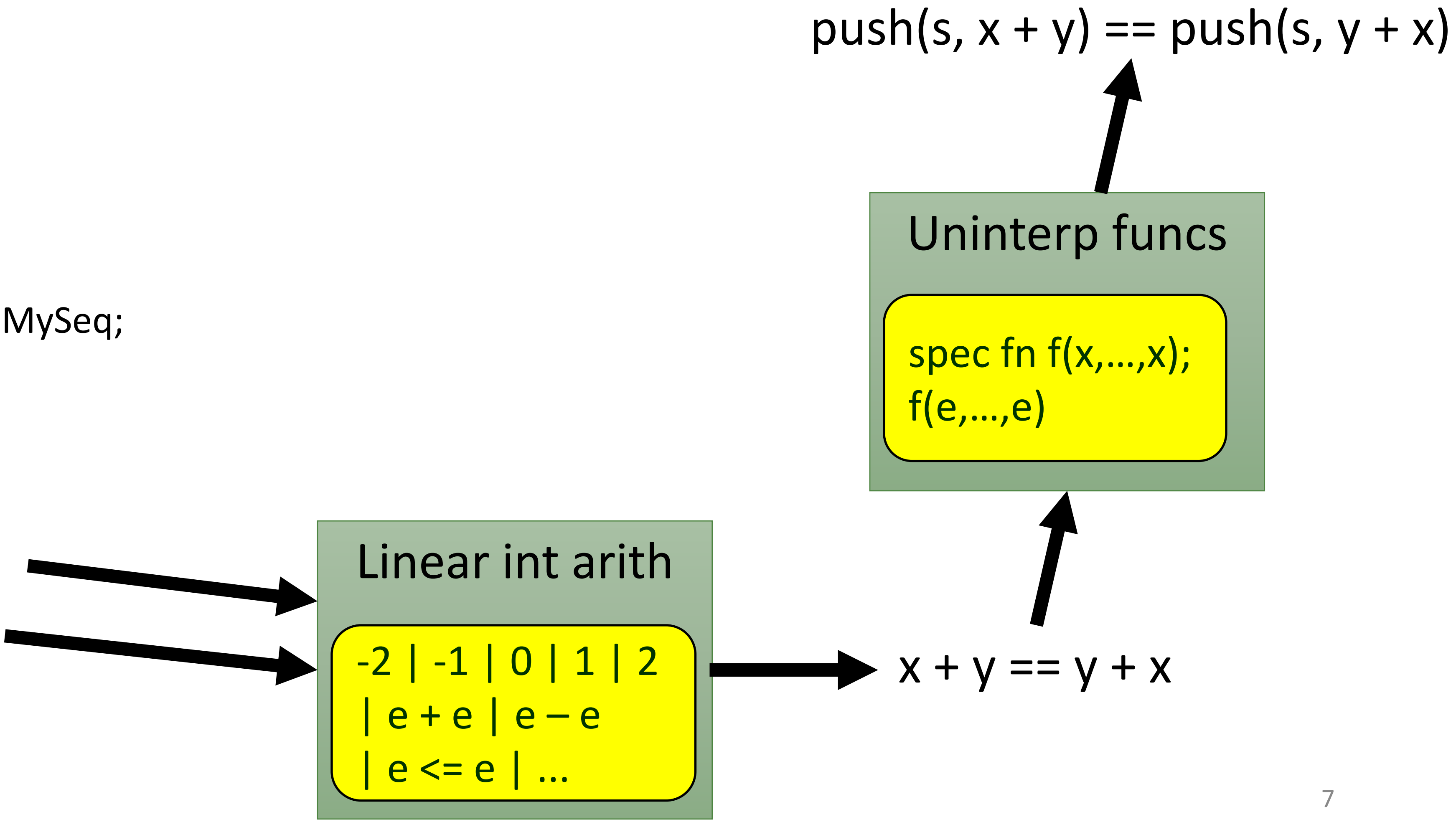
$$\left.\begin{array}{r} 2*x + y <= z \\ 3*x + 2*y <= z \\ z <= 4*x + 3*y \\ z <= 5*x + 4*y \end{array}\right\} \Rightarrow \begin{array}{l} 2*x + y <= 4*x + 3*y \\ 2*x + y <= 5*x + 4*y \\ 3*x + 2*y <= 4*x + 3*y \\ 3*x + 2*y <= 5*x + 4*y \end{array}$$

Sufficient for real numbers, rational numbers.

Integers require additional work.

Example: uninterp funcs + arithmetic

```
#[verifier::external_body]
struct MySeq;

impl MySeq {
    spec fn empty() -> MySeq;
    spec fn len(self) -> nat;
    spec fn push(self, value: int) -> MySeq;
}

proof fn test(x: int, y: int) {
    let s = MySeq::empty();
    assert(s.push(x + y)
        == s.push(y + x));
}
```



Exercise: uninterp funcs + arithmetic

```
impl MySeq {
    spec fn empty() -> MySeq;
    spec fn len(self) -> nat;
    spec fn push(self, value: int) -> MySeq;
}

proof fn axiom_my_seq_empty()
    ensures … empty len is 0 …
{ admit(); }

proof fn axiom_my_seq_push_len(s: MySeq, value: int)
    ensures … push adds 1 to len …
{ admit(); }

proof fn test(x: int, y: int) {
    let s0 = MySeq::empty();
    let s1 = s0.push(x + y);
    let s2 = s1.push(x - y);
    assert(s2.len() == 2); // make this succeed
}
```

Quantifiers

Quantifiers

```
forall|x: t| e
exists|x: t| e
```

```
spec fn f(i: int) -> int;
spec fn g(i: int) -> int;

proof fn test()
   requires
      forall|x: int, y: int| f(x) > g(y)
   ensures
      f(g(7)) > g(3)
{
}
```

Quantifiers

Quantifiers

forall|x: t| e
exists|x: t| e

Example:

$\forall$f,g.

(forall|x: int, y: int| f(x) > g(y))

==> f(g(7)) > g(3)

Quantifiers

Quantifiers

forall|x: t| e
exists|x: t| e

Example (as satisfiability query):
$\exists$f,g.
(forall|x: int, y: int| f(x) > g(y))
&& f(g(7)) <= g(3)

Infinitely many possible instantiations of x, y

…
&& f(2) > g(7)
&& f(4) > g(6)
&& f(4) > g(7)
&& f(5) > g(10)
&& f(5) > g(-5)
…

Quantifiers

Quantifiers

```
forall|x: t| e
exists|x: t| e
```

Example:

$$\exists f,g.\ \underbrace{}_{}$$

pattern (“trigger”) to match

```
∃f,g.
      (forall|x: int, y: int| #![trigger f(x), g(y)] f(x) > g(y))
  && f(g(7)) <= g(3)
```

Quantifiers

Quantifiers

forall|x: t| e
exists|x: t| e

Example: (alternate trigger notation for trigger f(x), g(y))

∃f,g.

(forall|x: int, y: int| #[trigger] f(x) > #[trigger] g(y))

&& f(g(7)) <= g(3)

Quantifiers

Quantifiers

forall|x: t| e
exists|x: t| e

Example:

pattern (“trigger”) to match

∃f,g.
(forall|x: int, y: int| #![trigger f(x), g(y)] f(x) > g(y))
&& f(g(7)) <= g(3)

matches f(x) with x = g(7)

matches g(y) with y = 3

Quantifiers

Quantifiers

forall|x: t| e
exists|x: t| e

Example:

pattern ("trigger") to match

∃f,g.

(forall|x: int, y: int| #![trigger f(x), g(y)] f(x) > g(y))

&& f(g(7)) <= g(3) && f(g(7)) > g(3)

matches f(x) with x = g(7)

matches g(y) with y = 3

contradiction
(no counterexample,
original formula is valid)

Quantifiers

Quantifiers

```
forall|x: t| e
exists|x: t| e
```

Example:

pattern (“trigger”) to match

```
∃f,g.
      (forall|x: int, y: int| #![trigger f(x), g(y)] f(x) > g(y))
  && f(g(7)) <= g(3)
```

Triggers may be written by user,
or chosen automatically by Verus.

Quantifiers

Quantifiers

```
forall|x: t| e
exists|x: t| e
```

Example:

pattern (“trigger”) to match

```
∃f,g.
      (forall|x: int, y: int| #![trigger f(x), g(y)] f(x) > g(y))
  && f(g(7)) <= g(3)
```

- The trigger must mention all the quantified variables
  - both and x and y
- The trigger matches only if *all* of its patterns match
  - both f(x) and g(y) must match some expressions

Quantifiers

Quantifiers

forall|x: t| e
exists|x: t| e

Example:

badly behaved trigger

∃f.
(forall|x: int| #![trigger f(x)] f(f(x)) > f(x)
&& f(3) <= f(f(2))
&& f(f(3)) > f(3)
&& f(f(f(3))) > f(f(3))
&& f(f(f(f(3)))) > f(f(f(3)))
…

Beware of infinite matching loops

Quantifiers

Quantifiers

forall|x: t| e
exists|x: t| e

nicely behaved trigger

```
Example:
∃f.
      (forall|x: int| #![trigger f(f(x))] f(f(x)) > f(x)
  && f(3) <= f(f(2))
  && f(f(2)) > f(2)
```

Triggers are the programming language for quantifiers.
Choose triggers carefully!

Exercise: quantifiers

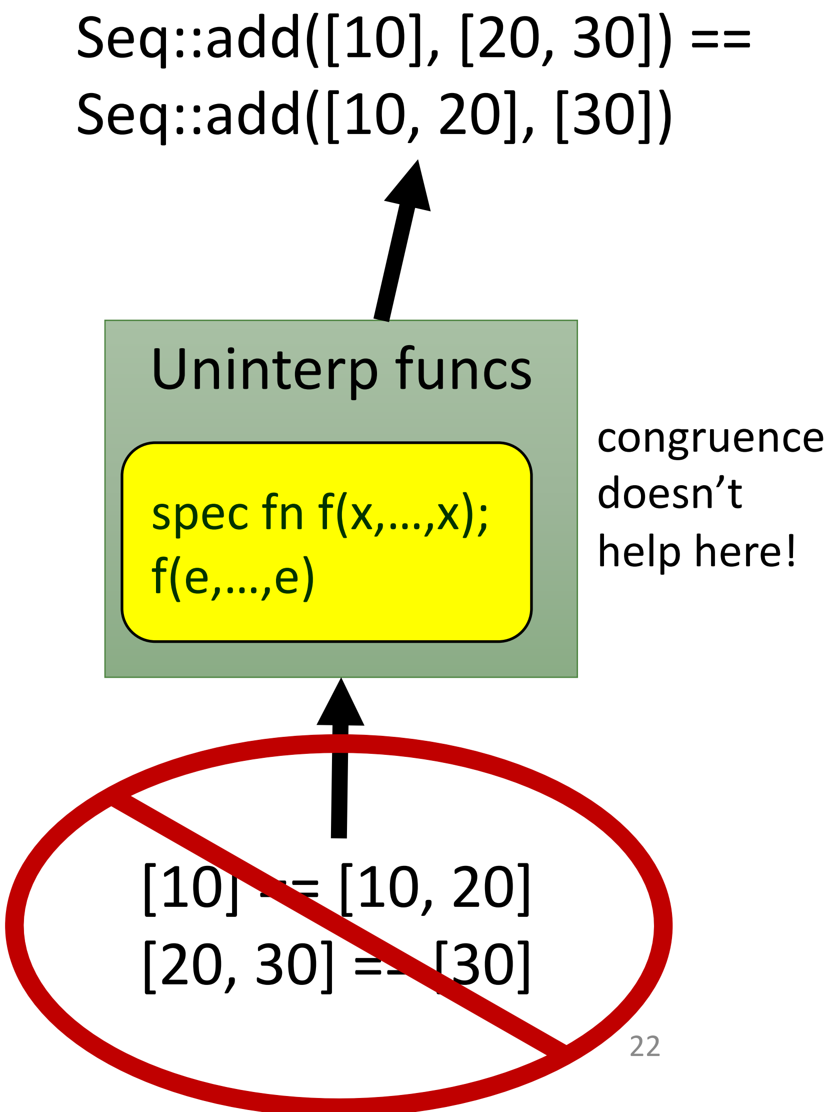
```
impl MySeq {
    spec fn empty() -> MySeq;
    spec fn len(self) -> nat;
    spec fn push(self, value: int) -> MySeq;
}

proof fn axiom_my_seq_empty()
    ensures … empty len is 0 …
{ admit(); }

proof fn axiom_my_seq_push_len_quant()
    ensures forall|s: MySeq, value: int| … push adds 1 to len …
{ admit(); }

proof fn test(x: int, y: int) {
    let s0 = MySeq::empty();
    let s1 = s0.push(x + y);
    let s2 = s1.push(x - y);
    assert(s2.len() == 2); // make this succeed
}
```

Exercise: broadcasts

```
broadcast proof fn axiom_my_seq_empty()
    ensures #[trigger] MySeq::empty().len() == 0
{ admit(); }

broadcast proof fn axiom_my_seq_push_len(s: MySeq, value: int)
    ensures … push adds 1 to len …
{ admit(); }

proof fn test(x: int, y: int) {
    broadcast use axiom_my_seq_empty;
    broadcast use axiom_my_seq_push_len;

    let s0 = MySeq::empty();
    let s1 = s0.push(x + y);
    let s2 = s1.push(x - y);
    assert(s2.len() == 2); // make this succeed
}
```

Exercise: extensional equality

```
proof fn axiom_seq_equal(x: Seq<u8>, y: Seq<u8>)
    requires
        …x and y’s lengths are equal…,
        …x and y’s elements are equal…,
    ensures
        x == y,
{ admit() }

proof fn demand_eq(x: Seq<u8>, y: Seq<u8>)
    requires x == y
{}

proof fn test_seq_eq() {
    axiom_seq_equal(…);
    demand_eq(seq![10] + seq![20, 30],
              seq![10, 20] + seq![30]); // make this succeed
}
```



Interpreted functions (nonrecursive)

Core Components (always enabled)

Uninterp funcs

spec fn f(x,…,x);
f(e,…,e)

Quantifiers

forall|x: t| e
exists|x: t| e

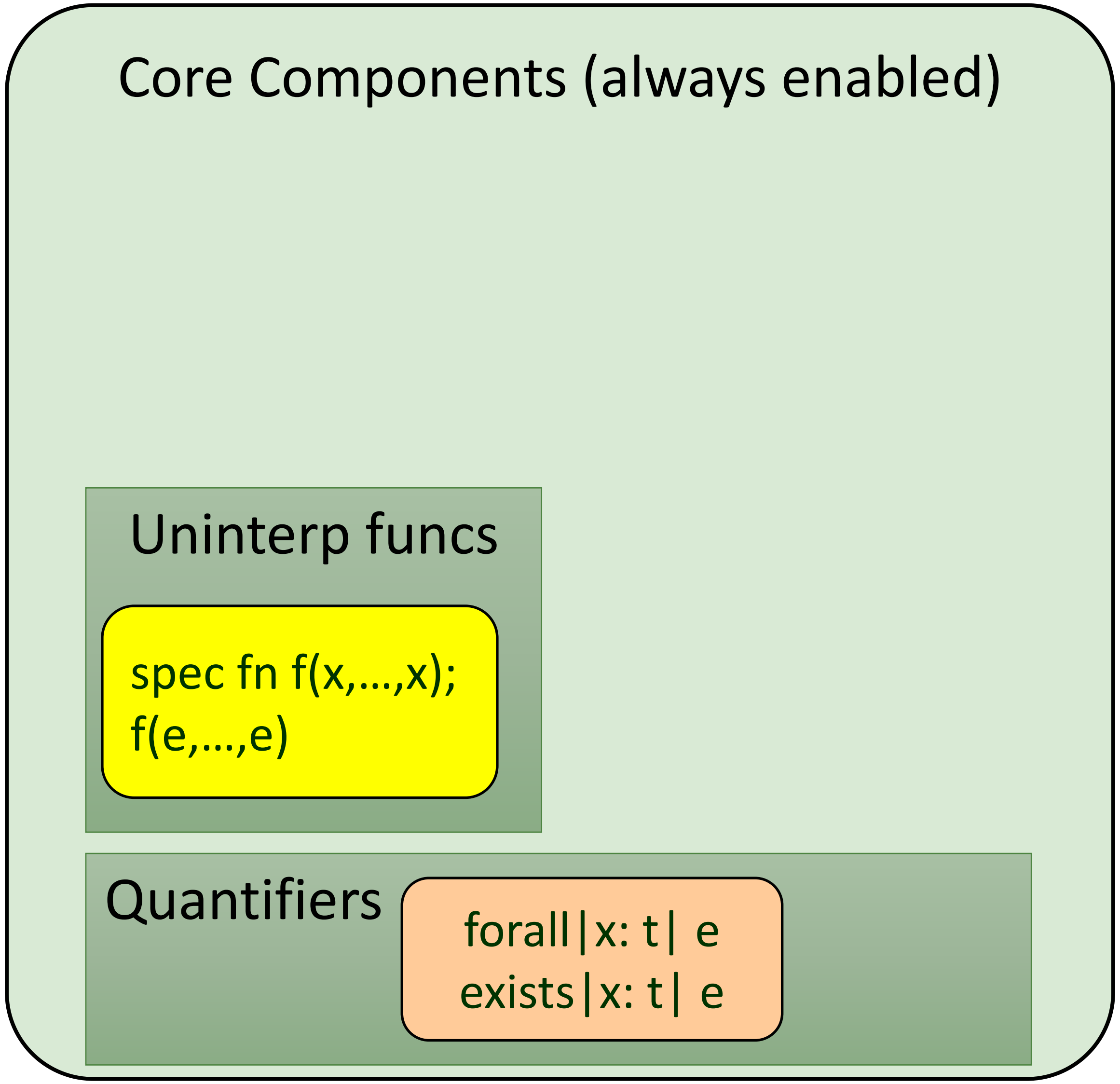
```
spec fn f(x: int, y: int) -> int {
    x + 2 * y
}
```

```
spec fn f(x: int, y: int) -> int;
```

axiom:

```
forall|x: int, y: int|
    #[trigger] f(x, y) == x + 2 * y
```

Interpreted functions (recursive)




```
spec fn f(x: int) -> bool
    decreases x
{
    x <= 0 || !f(x - 1)
}
```

Naive axiom generates matching loop:

```
      f(x)
   == x <= 0 || !f(x - 1)
   == x <= 0 || !(x – 1 <= 0 || !f(x - 1 - 1))
   == x <= 0 || !(x – 1 <= 0 || !(x – 1 – 1 <= 0 || !f(…)))
   …
```

Verus axiom uses "fuel" to limit unrolling depth

```
proof fn test() {
    reveal_with_fuel(f, 4);
    assert(!f(3));
}
```

Assert-by for bit vector, nonlinear arith

Core Components (always enabled)

SAT solver

true | false | !e
e && e | …

Linear int arith

-2 | -1 | 0 | 1 | 2
| e + e | e – e
| e <= e | …

Uninterp funcs

spec fn f(x,…,x);
f(e,…,e)

Datatypes

struct S { … }
enum E { … }
S(e,…,e) | e.f | …

Quantifiers

forall|x: t| e
exists|x: t| e

Additional Components (via “assert by”)

Bit vector arith

e & e| e << e |
…

Nonlin int arith

e * e
e / e | e % e

```
proof fn test_by(x: u32)
    requires x <= 200,
{
    assert(x >> 8 == 0) by(bit_vector)
        requires(x <= 255);
}
```

Assert-by for bit vector, nonlinear arith

Core Components (always enabled)

SAT solver
true | false | !e
e && e | …

Linear int arith
-2 | -1 | 0 | 1 | 2
| e + e | e – e
| e <= e | …

Uninterp funcs
spec fn f(x,…,x);
f(e,…,e)

Datatypes
struct S { … }
enum E { … }
S(e,…,e) | e.f | …

Quantifiers
forall|x: t| e
exists|x: t| e

Additional Components (via “assert by”)

Bit vector arith
e & e| e << e |
…

Nonlin int arith
e * e
e / e | e % e

```
proof fn test_by(x: u32, y: u32)
    requires x > 1 && y > 0
{
    assert(x * y > y) by(nonlinear_arith)
        requires(x > 1 && y > 0);
}
```