



Efficiently Tracking Floating Point Exceptions in High Performance Applications

Adam Brohl
Department of Computer Science
University of Rochester
October 2026



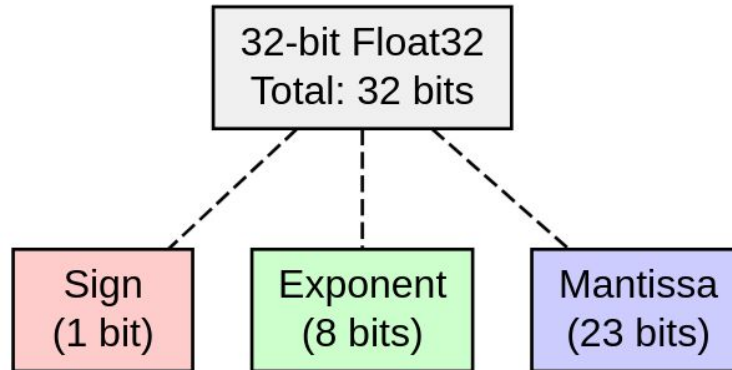
Outline

- Introduction to FP Checking
- Organizing Checks into a Decision Tree
- Formalizing Decision Tree Construction in Lean
- Expanding the Decision Tree to Check Multiple Floats Simultaneously
- Optimizing Non-exceptional paths in Decision Trees

Introduction to FP Checking and Why it is Important

Floating Point Numbers

- Floating point numbers are an approximation of real numbers.
- Cover a large range of numbers to cover as many applications as possible.





Exceptional Values

- Operations on floating point values may sometimes result in exceptional values
 - Positive Infinity
 - Negative Infinity
 - NaN
- *Example:*
 - $1 \div 0 = \infty$
 - $\text{MAX_FLOAT} + \text{MAX_FLOAT} = \infty$
 - $\infty - \infty = \text{NaN}$



Exceptional Values Propagate

- Exceptional values are exceptional because they represent invalid values.
 - Positive/Negative Infinity: Very large or small quantities.
 - NaN: A completely nonsensical value.
- Operations on exceptional values (almost) always produce more exceptional values.
 - $5 + \infty = \infty$
 - $\text{NaN} + y = \text{NaN}$



Real World FP Errors - Anthropic 2024

- Different precision floats used incorrectly (bf16 vs fp32)
- Affected highest probability token selection
- Led to wildly inaccurate responses generated by Claude.



Real World FP Errors - Patriot Missile Failure

- Interception missile failed to intercept target and hit US army barracks
- Lead to the death of 28 US soldiers
- Caused by a rounding error which got worse over time
- $0.1 = 0.0001100110011001101\dots$ (repeats forever)
- Fixed point, but similar issues exist with floating point



What are FP Checks

- Functions that record information of FP operation results
 - Is this number NaN?
 - Could this number overflow?
 - etc.
- Inserted after *every* operation
 - After compile time

```
Float x = a + b;
```

```
+ check(x);
```

```
Float y = x * 2 + c;
```

```
+ check(x * 2);
```

```
+ check(y);
```



FPChecker

- Dynamic analysis tool to identify FP exceptions in applications
- Operates on C/C++ Code
- Inserts checks at every floating point instruction
- Compiles a histogram detailing check information

Laguna, Ignacio. "Fpchecker: Detecting floating-point exceptions in gpu applications." In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1126-1129. IEEE, 2019.

FP Decision Trees


$$x = y \text{ `op` } z$$

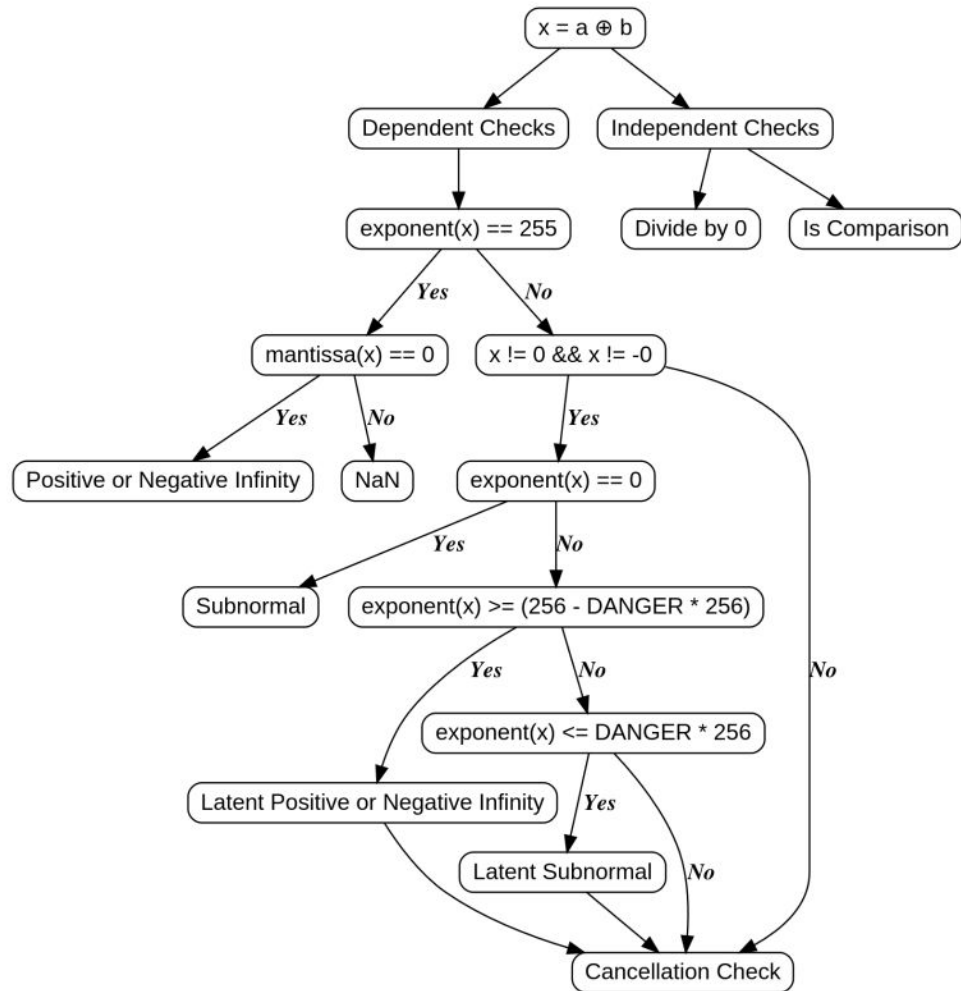
FPChecker Checks

```
item.infinity_pos = (uint64_t)_FPC_FP64_IS_INFINITY_POS(x);
item.infinity_neg = (uint64_t)_FPC_FP64_IS_INFINITY_NEG(x);
item.nan = (uint64_t)_FPC_FP64_IS_NAN(x);
item.division_zero = (uint64_t)_FPC_FP64_IS_DIVISION_ZERO(y, z, op);
item.cancellation = (uint64_t)_FPC_FP64_IS_CANCELLATION(x, y, z, op);
item.comparison = (uint64_t)_FPC_FP64_IS_COMPARISON(op);
item.underflow = (uint64_t)_FPC_FP64_IS_SUBNORMAL(x);
item.latent_infinity_pos = (uint64_t)_FPC_FP64_IS_LATENT_INFINITY_POS(x);
item.latent_infinity_neg = (uint64_t)_FPC_FP64_IS_LATENT_INFINITY_NEG(x);
item.latent_underflow = (uint64_t)_FPC_FP64_IS_LATENT_SUBNORMAL(x);
```



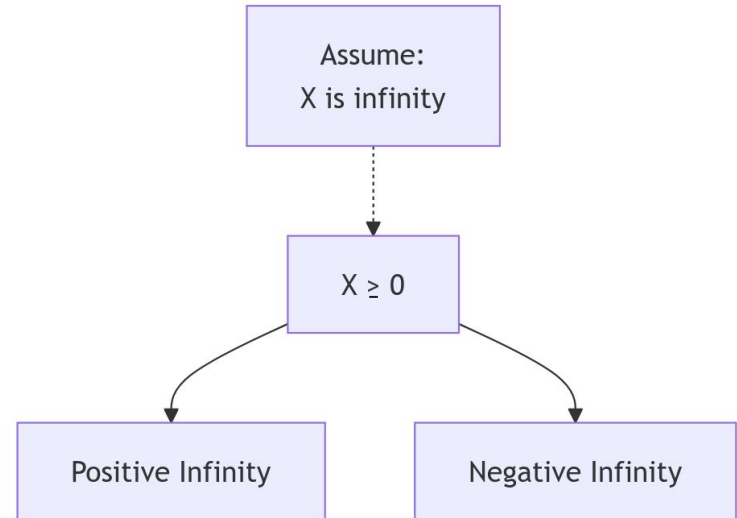
Mutually Exclusive Checks

- Some checks reveal information about others
 - Exponent == 255 -> Exceptional value
 - Nan -> not positive or negative infinity
- Some checks are mutually exclusive
 - Positive infinity
 - Negative infinity
- Checks can be structured into a hierarchy



Binary Decision Trees

- Each node (predicate) has two branches
 - True branch
 - False branch
- Mutually exclusive checks appear on separate branches
- Allows for an arbitrary set of checks





Creating a Binary Decision Tree

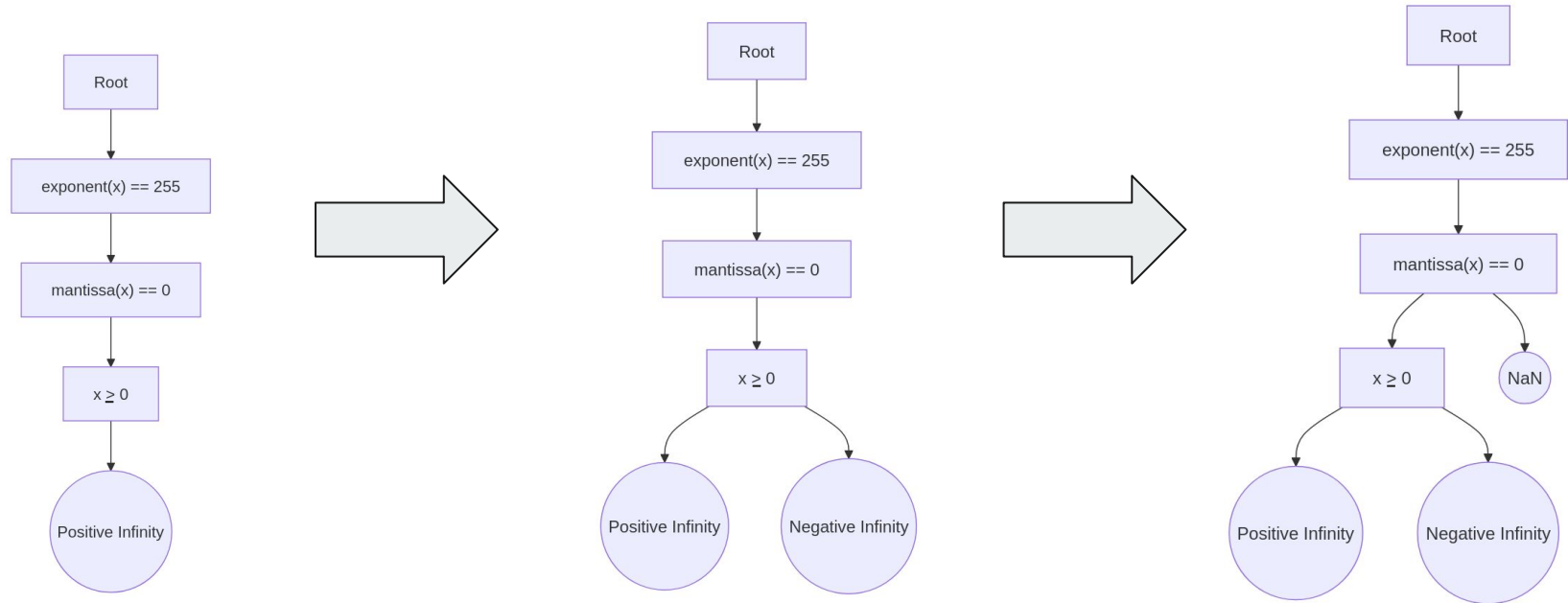
- Conditions are represented by pure functions
 - Input: float
 - Output: boolean
- Mutual exclusion can be checked with an SMT solver
 - Example: $\text{posinf}(x) \ \& \ \text{neginf}(x)$ is unsat



Overview of the Tree Construction Algorithm

- Split each check into a list of predicates joined by AND.
- Build a tree with the first check's predicates in the list.
- For all other checks in the list:
 - Traverse each predicate alongside the current tree
 - Use an SMT solver to determine exclusivity
 - Depending on the solver's output, propagate along the correct branch

High Level Overview of the Algorithm - Visualization



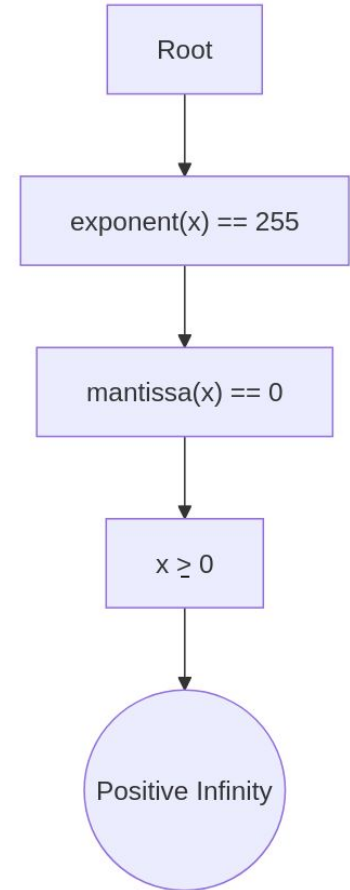


Constructing a Tree - Example

- **Positive Infinity:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) == 0$ and $x \geq_{\text{FP}} 0$
- **Negative Infinity:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) == 0$ and $x <_{\text{FP}} 0$
- **NaN:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) != 0$

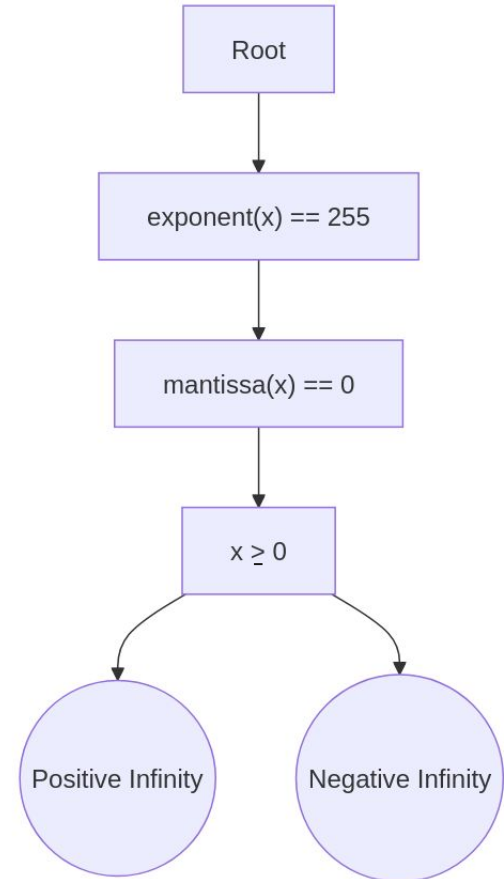
Constructing a Tree - Example

- **Positive Infinity:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) == 0$ and $x \geq_{\text{FP}} 0$



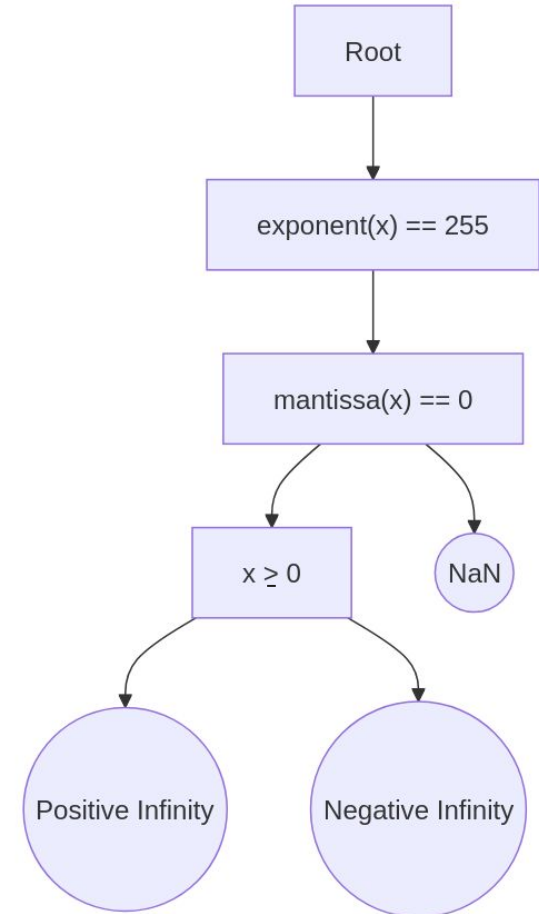
Constructing a Tree - Example

- **Positive Infinity:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) == 0$ and $x \geq_{\text{FP}} 0$
- **Negative Infinity:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) == 0$ and $x <_{\text{FP}} 0$



Constructing a Tree - Example

- **Positive Infinity:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) == 0$ and $x \geq_{\text{FP}} 0$
- **Negative Infinity:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) == 0$ and $x <_{\text{FP}} 0$
- **NaN:** $\text{exponent}(x) == 255$ and $\text{mantissa}(x) != 0$



Optimization Summary

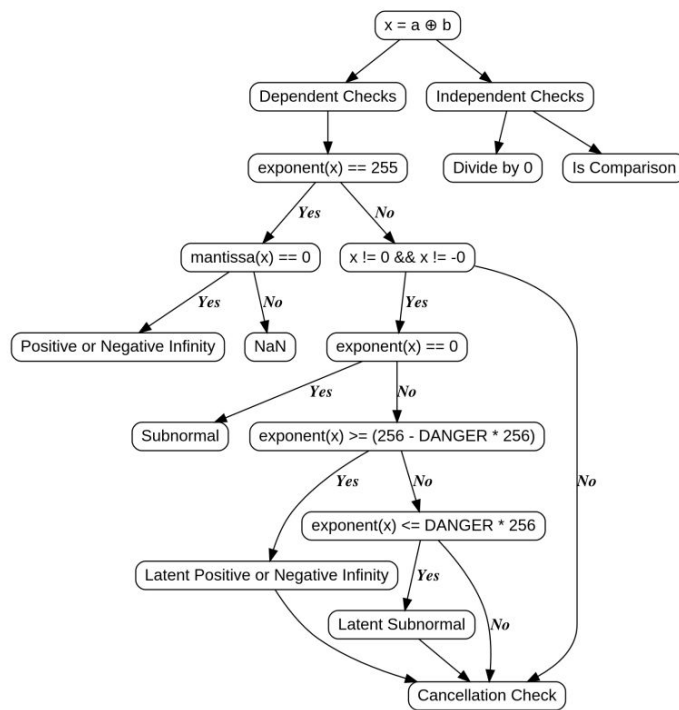
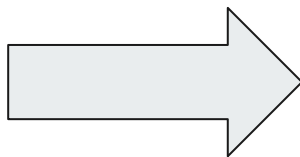
`check_posinf(...);`

`check_neginf(...);`

`check_nan(...);`

`check_subnormal(...);`

-
-
-



Formally Verifying Decision Tree Construction in Lean



How is Correctness Proven - Intro to Lean

- Proof Assistant
- Purely functional programming language with dependent types.
- Proofs written according to Curry-Howard Correspondence.
- Tactics for applying structurally applying transformations to proof inputs and outputs.

```
theorem de_morgan (p q : Prop)
  : ¬(p ∨ q) → ¬p ∧ ¬q := by
  intro h
  constructor
  . intro hp
    exact h (Or.inl hp)
  . intro hq
    exact h (Or.inr hq)
```

```
theorem de_morgan_no_tactics (p q : Prop)
  : ¬(p ∨ q) → ¬p ∧ ¬q :=
  λ pnq ↦ ⟨λ p ↦ pnq (Or.inl p), λ q ↦ pnq
    (Or.inr q)⟩
```



FP Tree DSL - Tree Representation

$$\begin{array}{lcl} DSLTree & ::= & \text{Leaf}(\text{FPCheck}+) \\ & | & \text{Node}(e_{\text{Bool}}, t_{lhs}, t_{rhs}) \end{array}$$
$$\text{FPCheck} ::= e_{\text{Bool}} +$$



FP Tree DSL

e_{Float}	$::=$	literal(v_{Float})
		inputVar
		binaryOp($op, e_{\text{Float}}, e_{\text{Float}}$)
		unaryOp(op, e_{Float})
e_{Int}	$::=$	literal(v_{Int})
		sign(e_{Float})
		exponent(e_{Float})
		significand(e_{Float})
		binaryOp($op, e_{\text{Int}}, e_{\text{Int}}$)
		unaryOp(op, e_{Int})
e_{Bool}	$::=$	literal(v_{Bool})
		is_equal($e_{\text{Float}}, e_{\text{Float}}$)
		not(e_{Bool})
		and($e_{\text{Bool}}, e_{\text{Bool}}$)
		or($e_{\text{Bool}}, e_{\text{Bool}}$)



Encoding in Lean

```
inductive DSLType where
| DSLFloat
| DSLBool

def from_dsl_type (ty : DSLType) : Type := match ty with
| DSLFloat => Float32
| DSLBool  => Bool

inductive DSL : DSLType -> Type 1 where
| literal : (value : from_dsl_type α) -> DSL α
| var      : DSL DSLFloat
| add      : Add (from_dsl_type α) -> DSL α -> DSL α -> DSL α
| is_equal : BEq (from_dsl_type α) -> DSL α -> DSL α -> DSL DSLBool
| not      : DSL DSLBool -> DSL DSLBool

abbrev FPCheck := List (DSL DSLBool)
```



Encoding in Lean - Decision Tree

```
inductive DSLTree where
  | Leaf : List FPCheck -> DSLTree
  | Node : DSL DSLBool -> DSLTree -> DSLTree -> DSLTree

def eval_tree (x : Float32) (t : DSLTree) : List FPCheck := match t with
| .Leaf cs => cs
| .Node p if_true if_false => match eval x p with
  | true => eval_tree x if_true
  | false => eval_tree x if_false
```



Solver Representation

Response	Proof Statement
Equivalent	$\forall x. \text{eval}(x, e_1) = \text{eval}(x, e_2)$
Exclusive	$\forall x. \text{eval}(x, e_1) \neq \text{eval}(x, e_2)$
Ambiguous	$\exists x. \text{eval}(x, e_1) = \text{eval}(x, e_2) \wedge \exists x. \text{eval}(x, e_1) \neq \text{eval}(x, e_2)$



Solver Representation

```
inductive SolverResponse (e1 e2 : DSL DSLBool) where
  | EquivCheck : (∀ x, eval x e1 = eval x e2) -> SolverResponse e1 e2
  | Exclusive : (∀ x, eval x e1 ≠ eval x e2) -> SolverResponse e1 e2
  | Ambiguous : (∃ x : Float32, eval x e1 = eval x e2) ∧
    (∃ x : Float32, eval x e1 ≠ eval x e2) -> SolverResponse e1 e2

abbrev Solver := (e1 e2 : DSL DSLBool) -> SolverResponse e1 e2
```

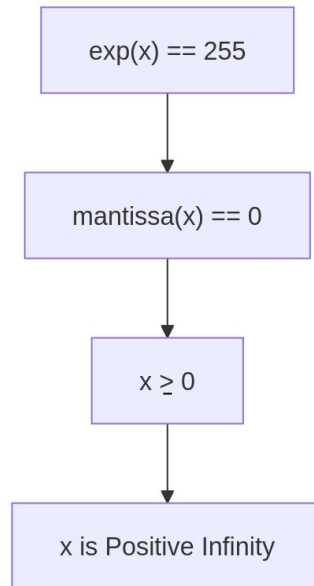
Proving the Initial Tree is Correct (When the Check is True)

```
pos_inf(x)
  = (exp(x) == 255)
  && (mantissa(x) == 0)
  && (x >= 0)
```

Assume `pos_inf(x)` is true.

Thus, all predicates must be true

Thus an evaluation of the tree traverses all true branches to **x is Positive Infinity**



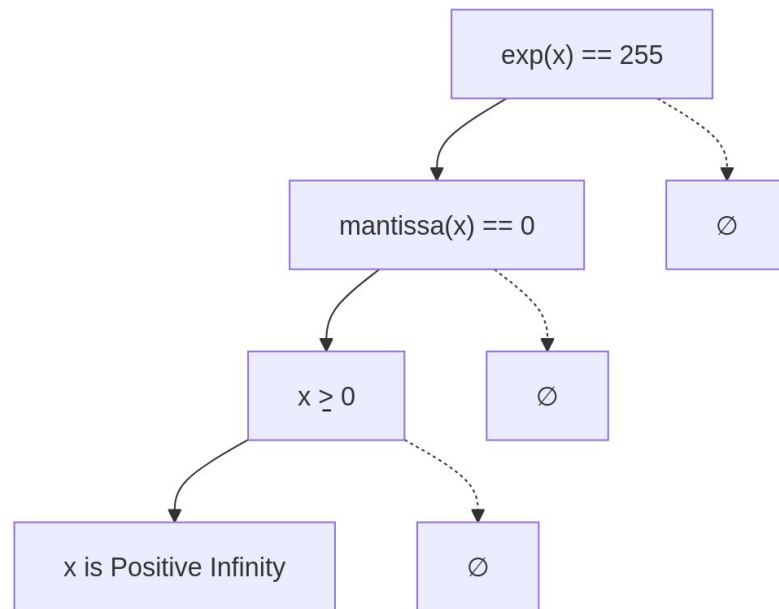
Proving the Initial Tree is Correct (When the Check is False)

```
pos_inf(x)
= (exp(x) == 255)
  && (mantissa(x) == 0)
  && (x >= 0)
```

Assume $\text{pos_inf}(x)$ is false.

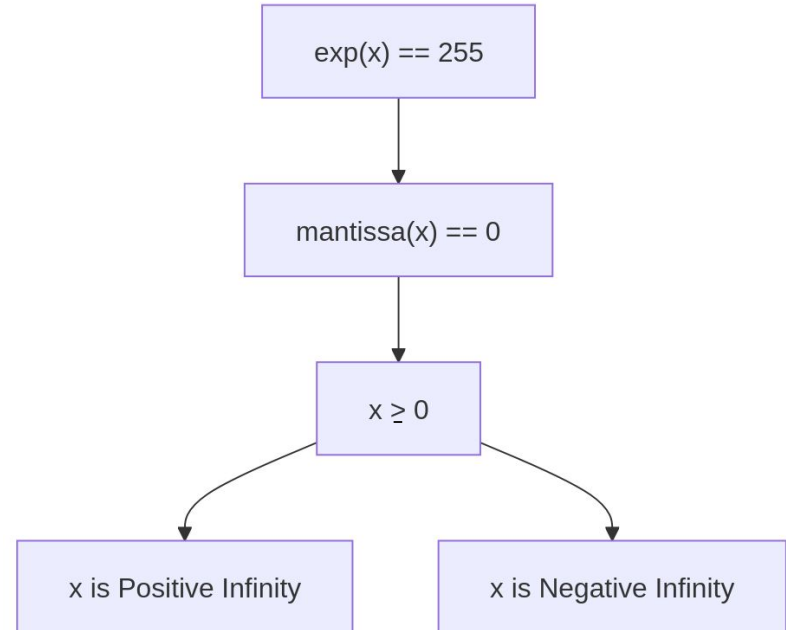
Thus, at least one predicate is false.

Thus evaluating the tree traverses at least one false path to $\emptyset$



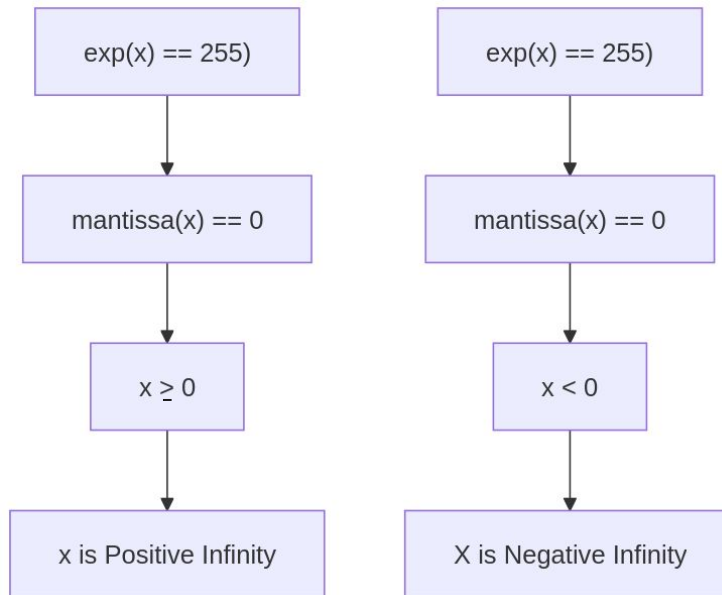
Proving Addition of Check - When the Check is True

- All predicates are true
- Solver shows which path to take
- Check shows up on its “true” paths



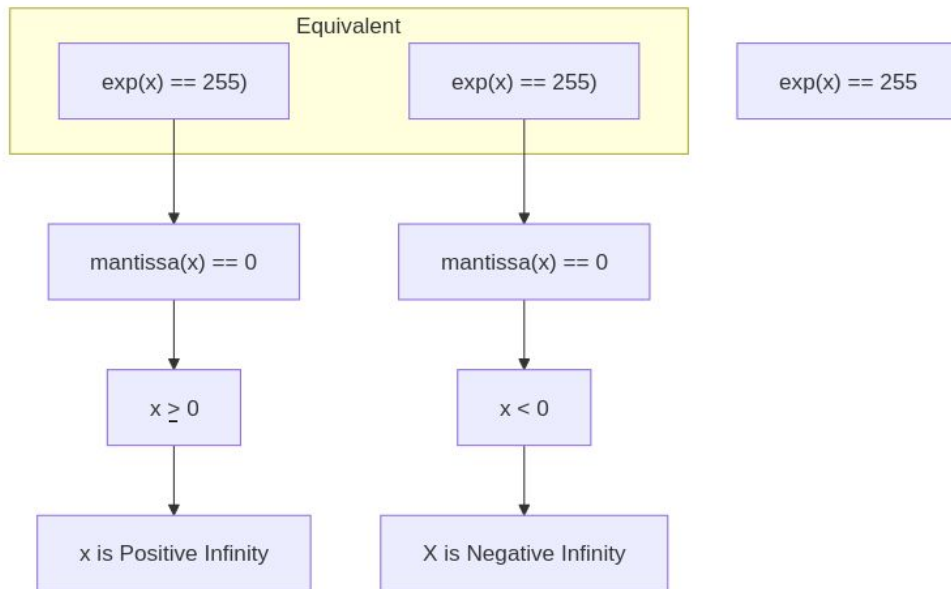
Proving the Addition of a Check is Correct (When x is Exceptional)

Recall: The solver returns proofs!



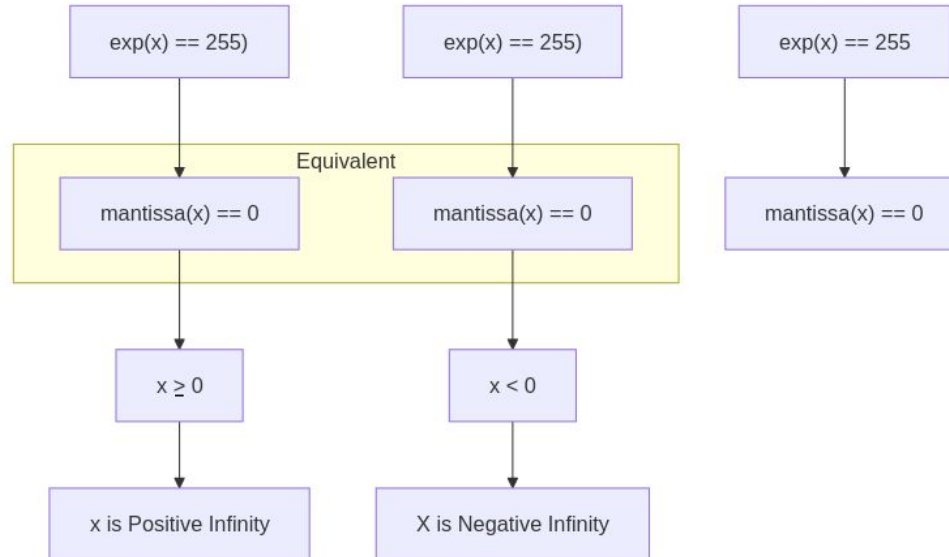
Proving the Addition of a Check is Correct (When x is Exceptional)

Recall: The solver returns proofs!



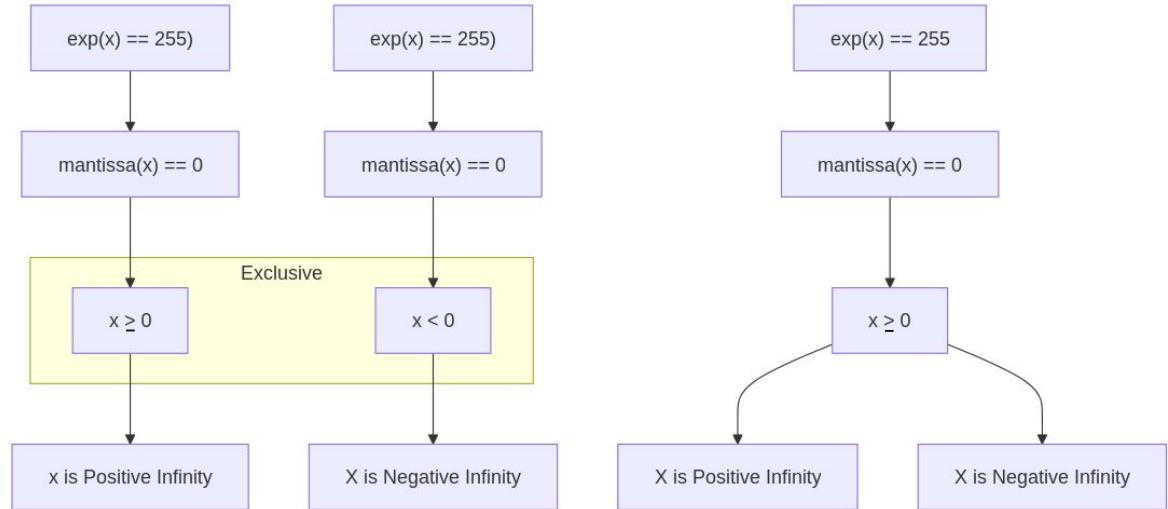
Proving the Addition of a Check is Correct (When x is Exceptional)

Recall: The solver returns proofs!



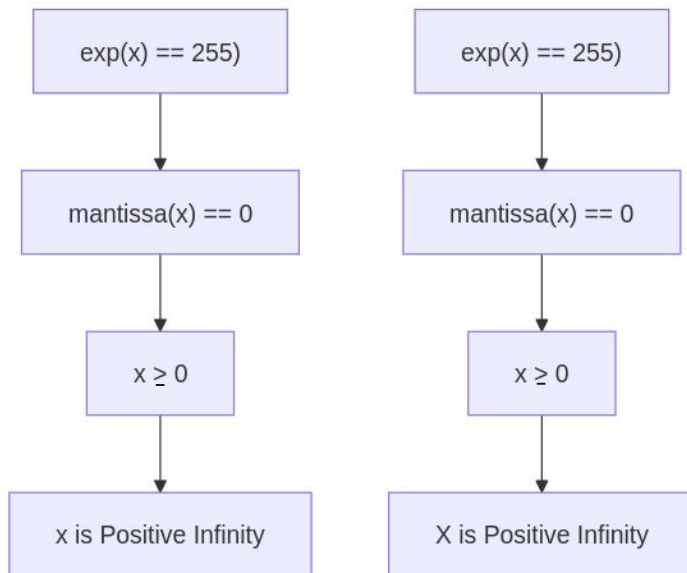
Proving the Addition of a Check is Correct (When x is Exceptional)

Recall: The solver returns proofs!



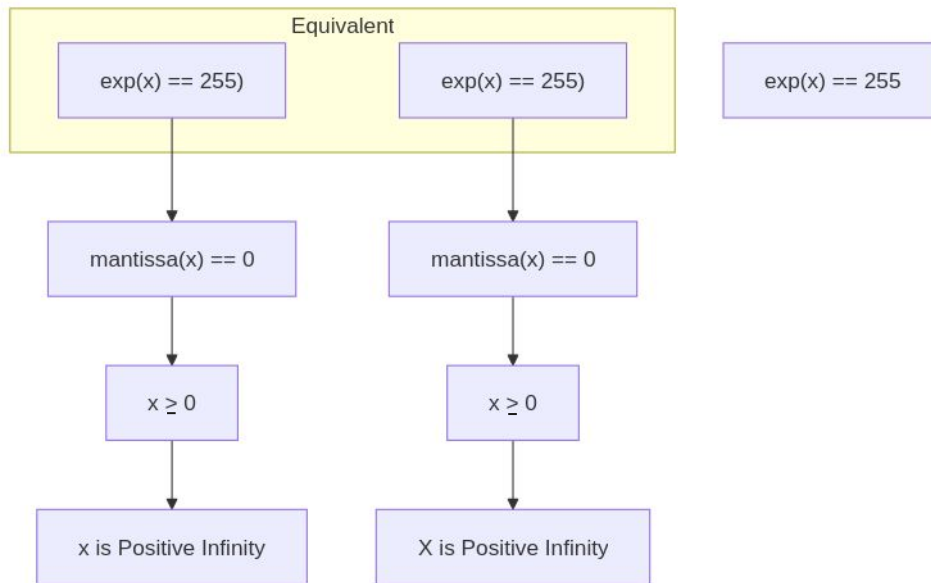
Proving Duplicate Checks are Redundant

- All predicates are true
- All new predicates should be skipped
- Solver will always return equivalent.



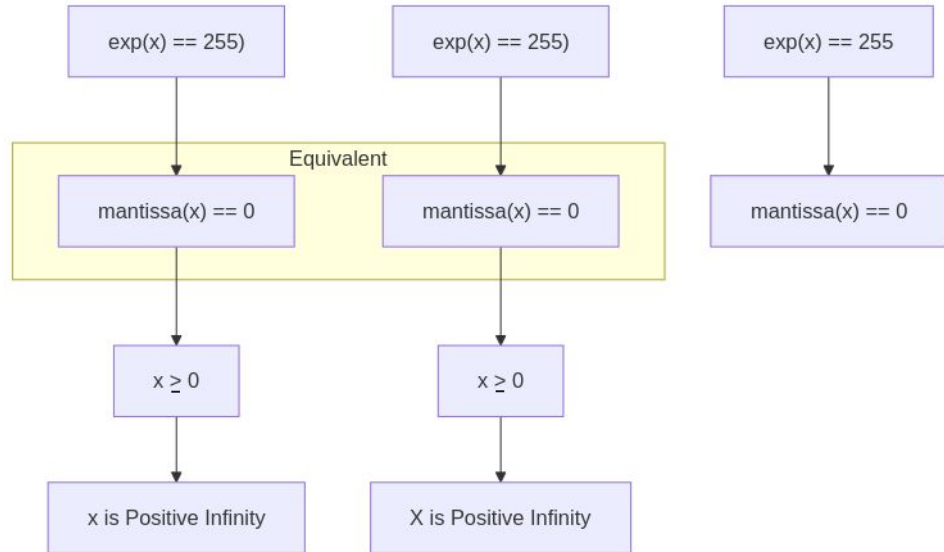
Proving Duplicate Checks are Redundant

- All predicates are true
- All new predicates should be skipped
- Solver will always return equivalent.



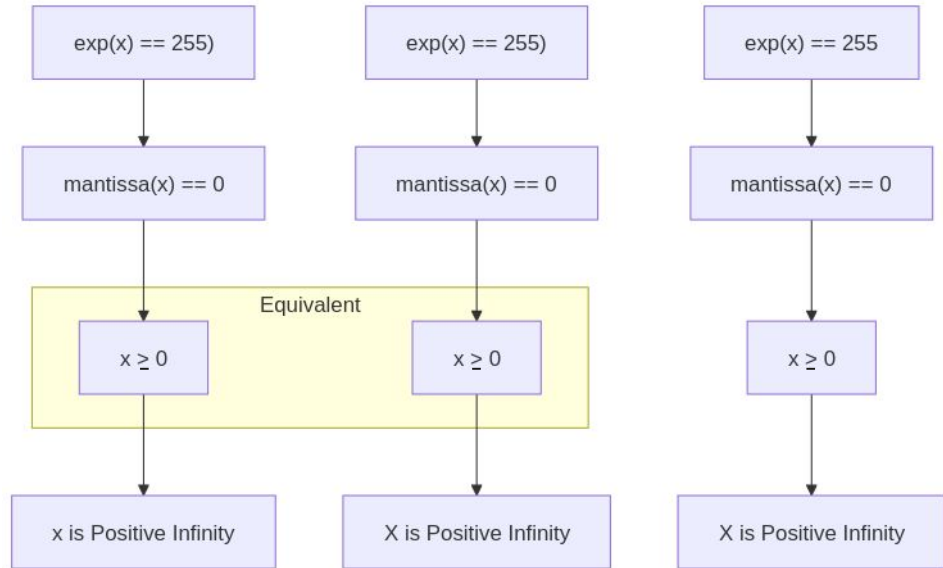
Proving Duplicate Checks are Redundant

- All predicates are true
- All new predicates should be skipped
- Solver will always return equivalent.



Proving Duplicate Checks are Redundant

- All predicates are true
- All new predicates should be skipped
- Solver will always return equivalent.



Why Solver is Guaranteed to return Equivalent

Response	Proof Statement
Equivalent	$\forall x. \text{eval}(x, e_1) = \text{eval}(x, e_2)$
Exclusive	$\forall x. \text{eval}(x, e_1) \neq \text{eval}(x, e_2)$
Ambiguous	$\exists x. \text{eval}(x, e_1) = \text{eval}(x, e_2) \wedge \exists x. \text{eval}(x, e_1) \neq \text{eval}(x, e_2)$

Checking Multiple Floats Simultaneously



Checking Multiple Values

- $Y = (A + B) + C$
 - If A is positive infinity, then Y is either positive infinity or NaN
 - If Y is positive Infinity, then either A + B or C is positive infinity.
- Checks of a value can imply checks of itself.
- Checks of a value can imply checks of other values.

Given TRUE

	$+\infty$	$-\infty$	NaN	Sub	Lat. $+\infty$	Lat. $-\infty$	Lat.Sub
$+\infty$	T	F	F	F	T	F	F
$-\infty$	F	T	F	F	F	T	F
NaN	F	F	T	F	F	F	F
Sub	F	F	F	T	F	F	T
Lat. $+\infty$	?	F	F	F	T	F	F
Lat. $-\infty$	F	?	F	F	F	T	F
Lat.Sub	F	F	F	?	F	F	T

Given FALSE

	$+\infty$	$-\infty$	NaN	Sub	Lat. $+\infty$	Lat. $-\infty$	Lat.Sub
$+\infty$	F	?	?	?	?	?	?
$-\infty$	?	F	?	?	?	?	?
NaN	?	?	F	?	?	?	?
Sub	?	?	?	F	?	?	?
Lat. $+\infty$	F	?	?	?	F	?	?
Lat. $-\infty$	?	F	?	?	?	F	?
Lat.Sub	?	?	?	F	?	?	F

Row = Operand
Column = Result of FAdd with Operand

Given TRUE

	$+\infty$	$-\infty$	NaN	Sub	Lat. $+\infty$	Lat. $-\infty$	Lat.Sub
$+\infty$	?	F	?	F	?	F	F
$-\infty$	F	?	?	F	F	?	F
NaN	F	F	T	F	F	F	F
Sub	?	?	?	?	?	?	?
Lat. $+\infty$	?	?	?	F*	?	?	?
Lat. $-\infty$	?	?	?	F*	?	?	F*
Lat.Sub	?	?	?	?	?	?	?

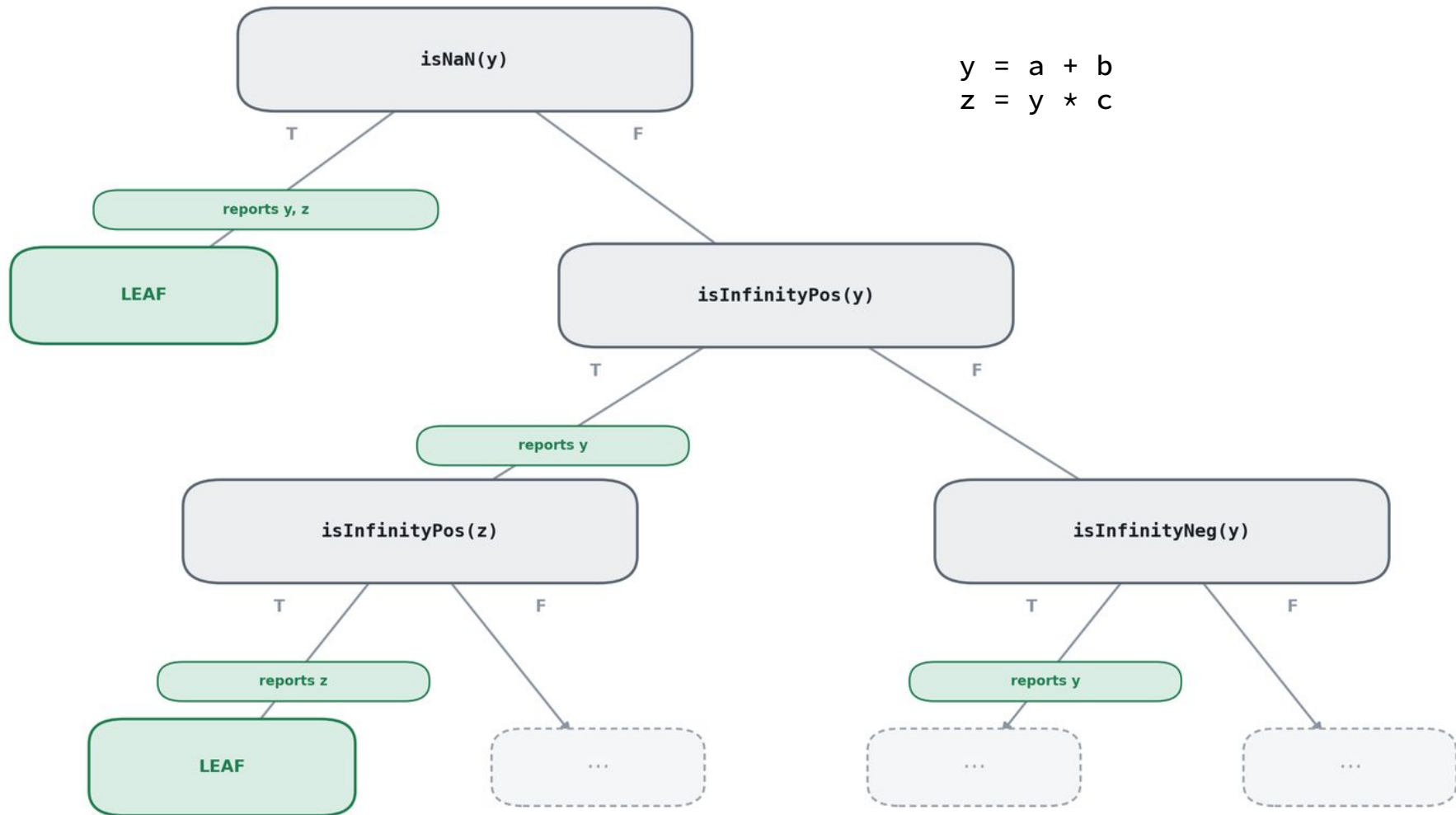
Given FALSE

	$+\infty$	$-\infty$	NaN	Sub	Lat. $+\infty$	Lat. $-\infty$	Lat.Sub
$+\infty$	?	?	?	?	?	?	?
$-\infty$	?	?	?	?	?	?	?
NaN	?	?	?	?	?	?	?
Sub	?	?	?	?	?	?	?
Lat. $+\infty$	?	?	?	?	?	?	?
Lat. $-\infty$	?	?	?	?	?	?	?
Lat.Sub	?	?	?	?	?	?	?



Tree Construction

- For each pair ($x \in \text{Float32}$, $c \in \text{FPCheck}$):
 - Find the pair with the highest amount of implications
 - Place at the top of the decision tree
 - Recursively apply until no checks are left



Short Circuiting on Non-exceptional Values



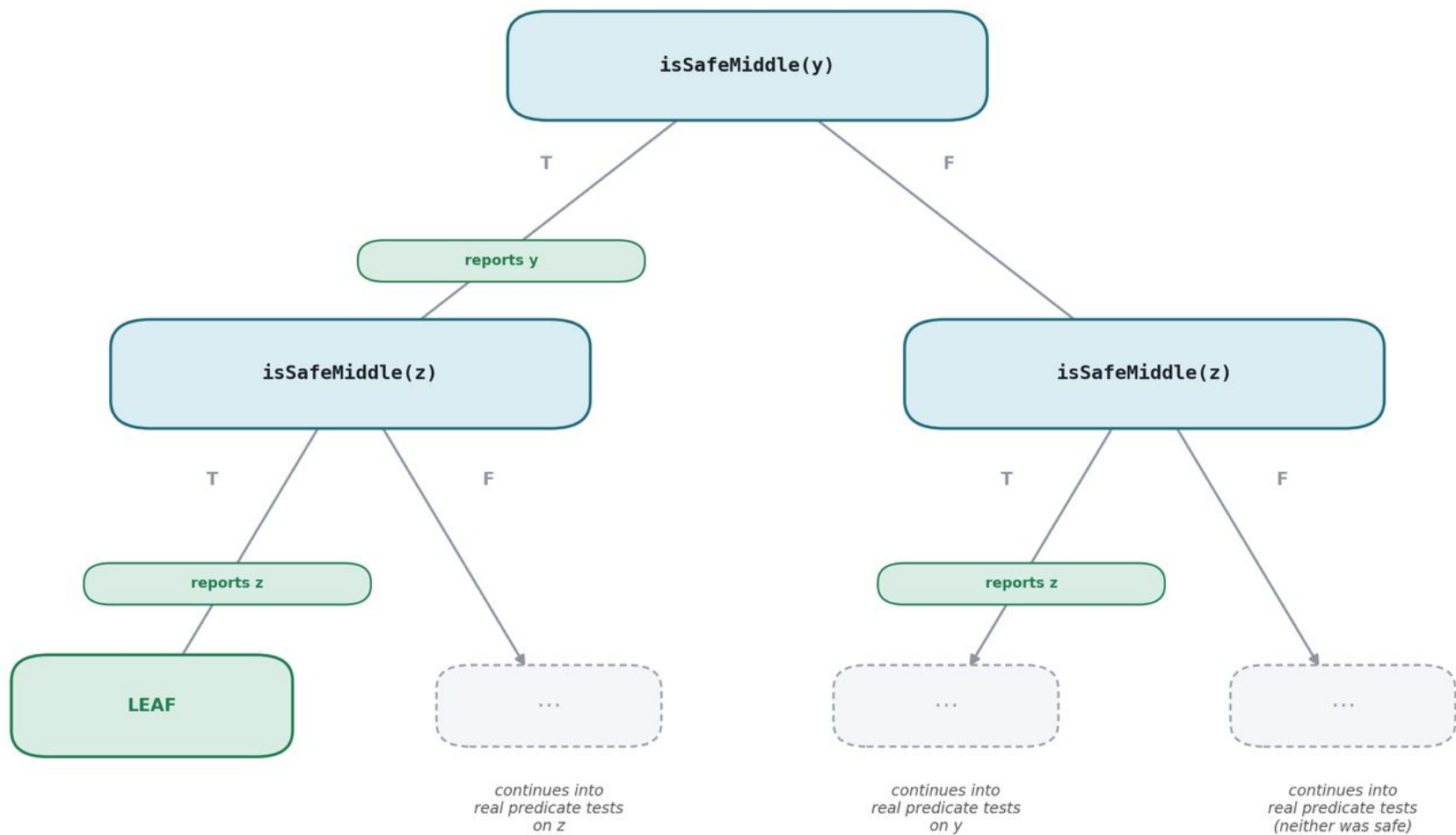
Handling Non Exceptional Values

- In most cases the result of an FP operation is a non-exceptional value.
- The decision tree primarily focuses on exceptional values
 - Non-exceptional values must traverse the entire tree
- Want to be able to quickly determine if a value is non exceptional or not.



Building a Check for Non Exceptional Values

- All the dependent checks focus on the exponent of a value.
- **Positive/Negative Infinity & NaN:** Check if exponent == 255
- **Latent Positive/Negative Infinity:** Check if exponent is above some threshold
- **Subnormal:** Check if exponent == 0
- **Latent Subnormal:** Check if exponent is below some threshold
- This provides a range of safe values and a cheap check





Automatically Generating SafeMiddle

- SafeMiddle is only efficient for some sets of FP Checks
- Not all checks fit the range approach
- Want to allow users to specify their own set of checks



SyGuS

- Syntax-Guided Program Synthesis
- User defines a grammar and post conditions of synthesized function
- A solver (e.g. CVC5) automatically generates candidates and checks if postconditions hold.



Synthesizing SafeMiddle

- Same DSL can be used for FP Checking and SafeMiddle
- Output is a function which is equivalent to `nonExceptional(x)`
- FP Checks are embedded as SMTLib functions
- `nonExceptional(x) = !pos_inf(x) && !neg_inf(x) && !is_nan(x) && ...`



Static Elimination

- Some checks only return true for some FP operations
 - Divide by zero: Nonsensical to run on FAdd or FMul
- No need to insert these checks when they are redundant
- Three operator dependent checks (from FPChecker)
 - Divide by Zero
 - Cancellation
 - Comparison



Comparing FP Check Counts to FPChecker

- FPChecker inserts 10 checks per operation
 - 3 can be statically removed for some operations
- Decision Tree (non-exceptional values):
 - **FAdd/FSub**: 2 checks (SafeMiddle + cancellation)
 - **FMul**: 1 check (SafeMiddle)
 - **FDiv**: 2 checks (SafeMiddle + divide by zero)

$$Y = (a * b) * c$$

- **FPChecker**: 20 checks
- **Decision Tree**: 2-16 Checks



Future Direction

- Weighted decision tree nodes
 - Some exceptional results are more common than others
 - Ex: positive infinity vs latent positive infinity
- Automated static elimination
 - Some checks are dependent on operation types
 - Ex: division by zero
- SafeMiddle synthesization with integer sets
 - Guarantee smallest expression
 - Set cardinalities can be used to determine weights for decision tree nodes