University of St.Gallen

Programming Group

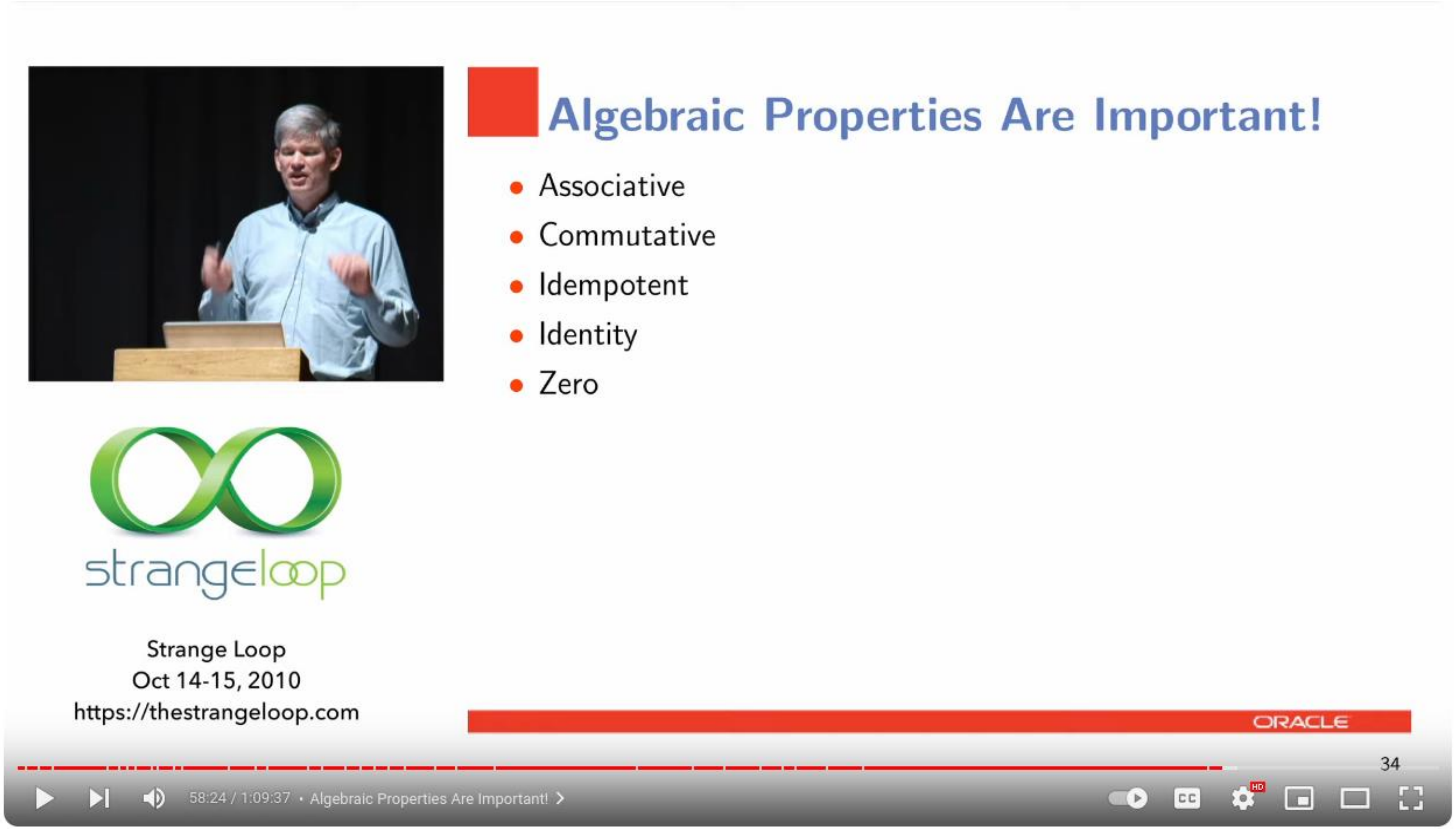
Propel

Verification of Algebraic Properties

George Zakhour

Pascal Weisenburger

Guido Salvaneschi



"How to Think about Parallel Programming: Not!" - Guy L. Steele Jr. (Strange Loop 2010)

Algebraic Properties are Everywhere

- Datastructure Invariants
- Compiler Optimization
- Stream Processing
- Algorithm Design
- Typeclass Laws
- Databases

Algebraic Properties are Everywhere

- Datastructure Invariants
    a. CRDTs
        - state-based (PLDI' 23)
        - op-based (PLF' 23)
- Compiler Optimization
- Stream Processing
- Algorithm Design
- Typeclass Laws
- Databases

University of St.Gallen

Programming Group

Propel

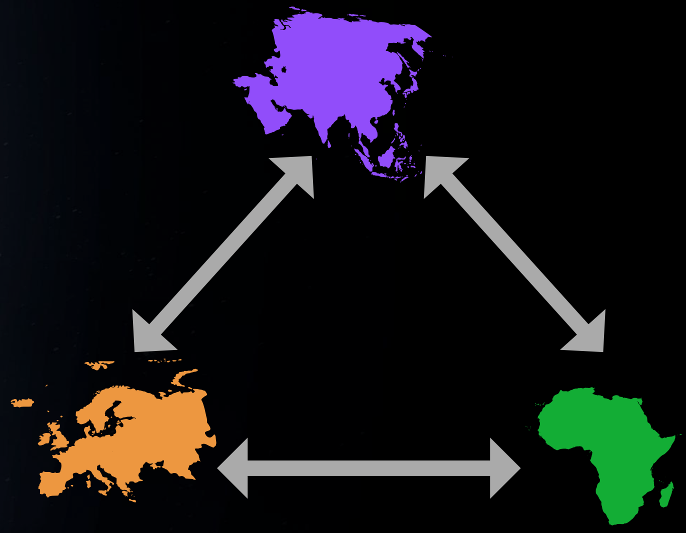
Type-Checking CRDT Convergence

George Zakhour

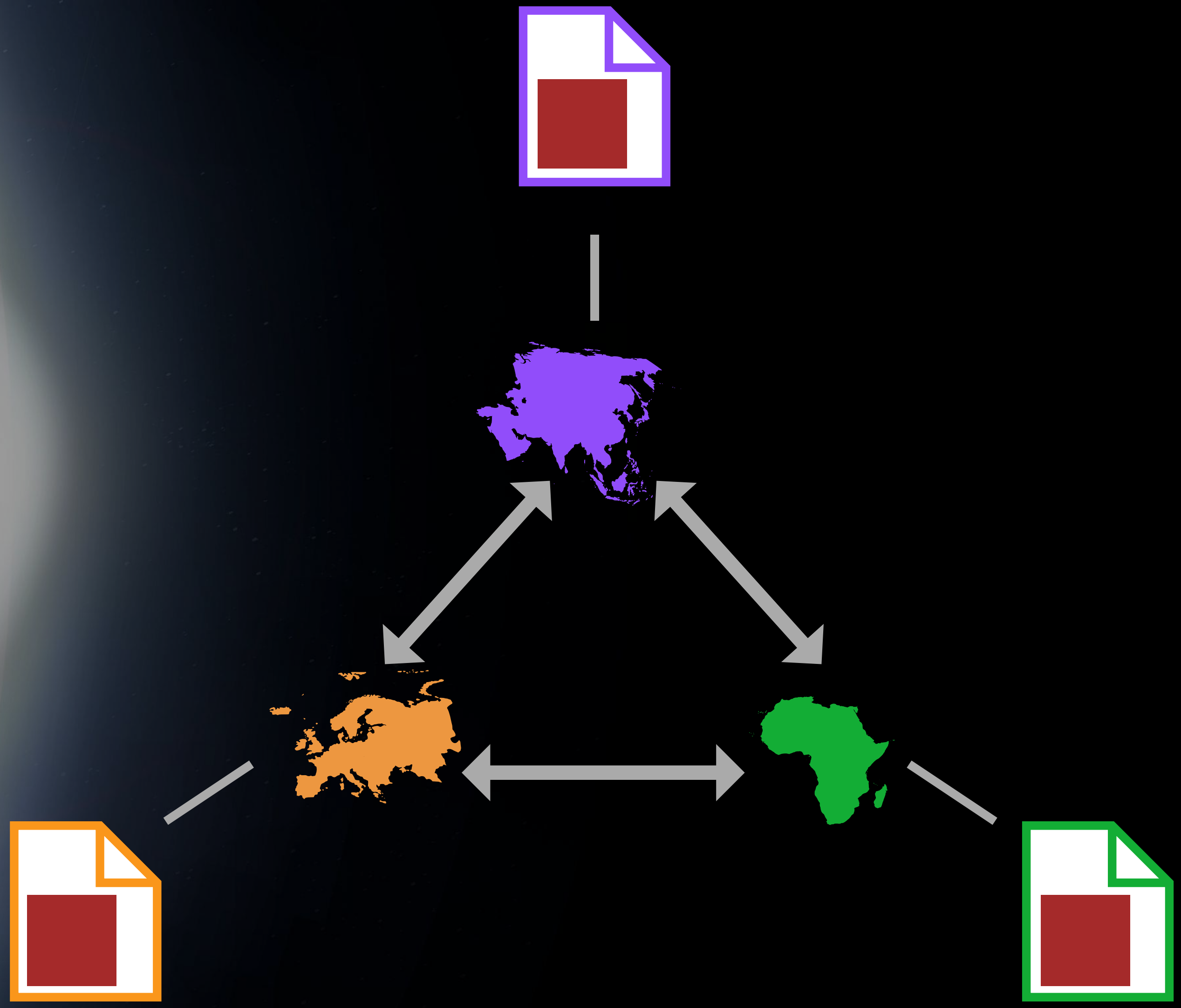
Pascal Weisenburger

Guido Salvaneschi

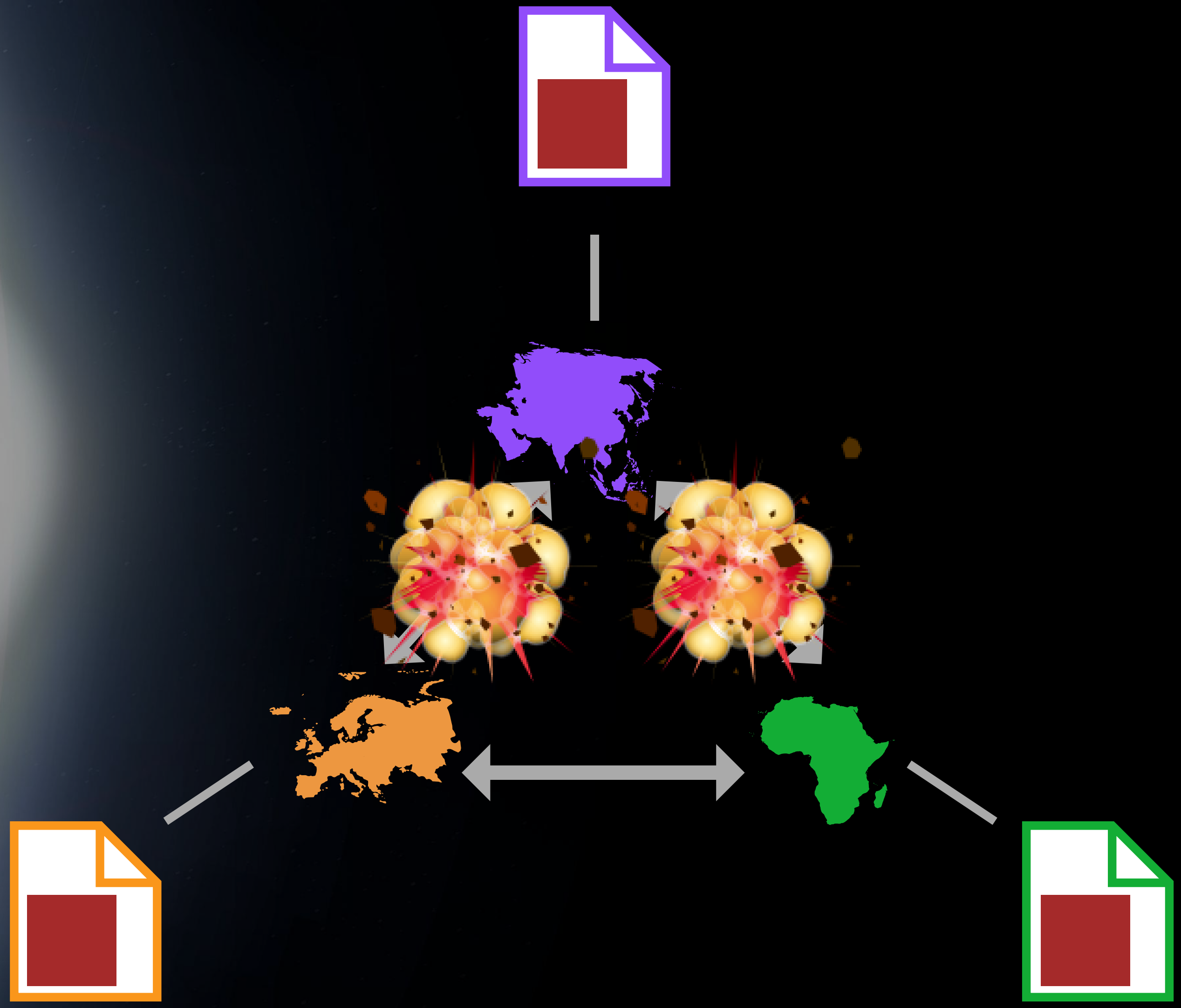
Distributed System



Distributed System: Replicated Data



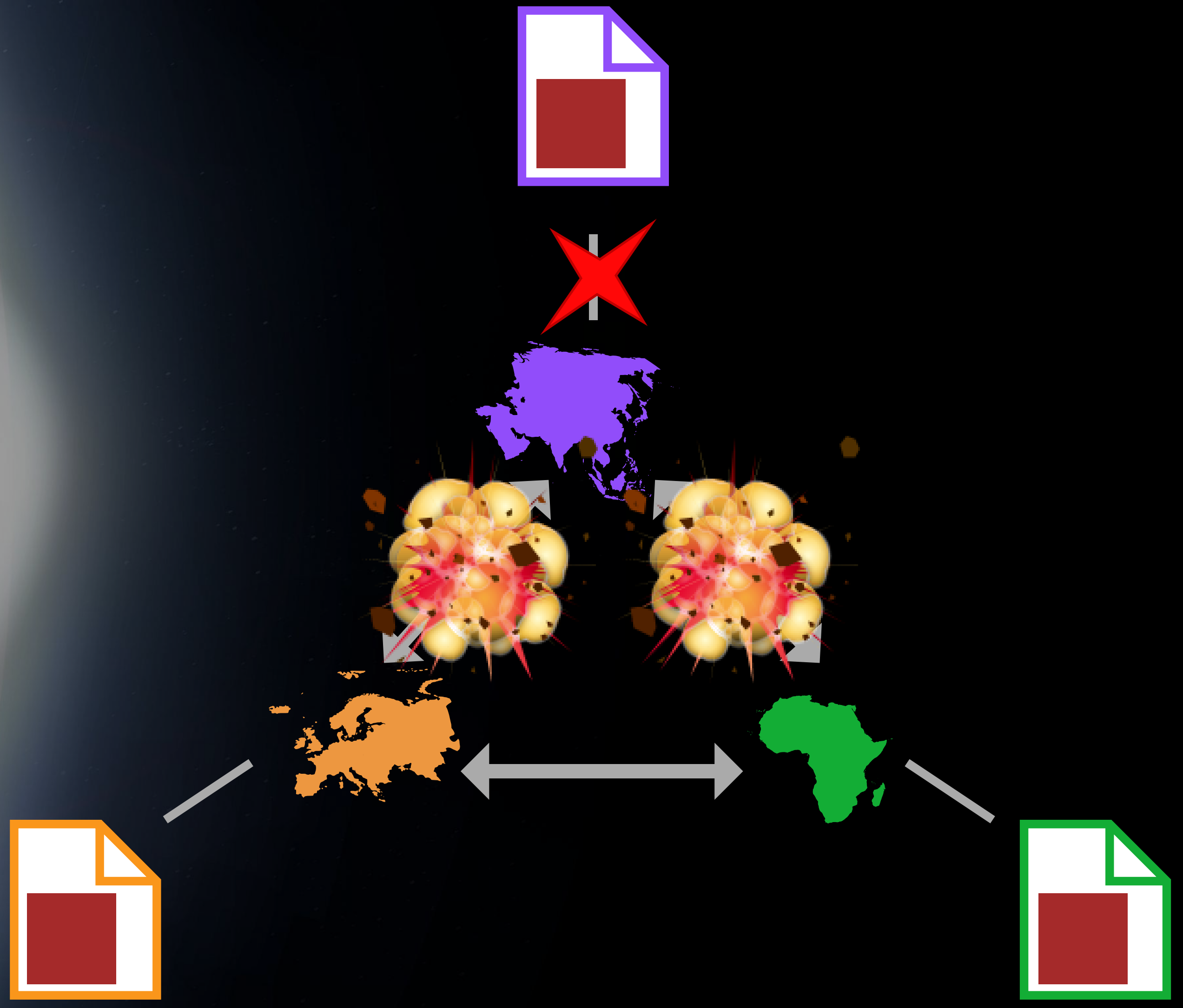
Distributed System: Replicated Data



Distributed System: Replicated Data

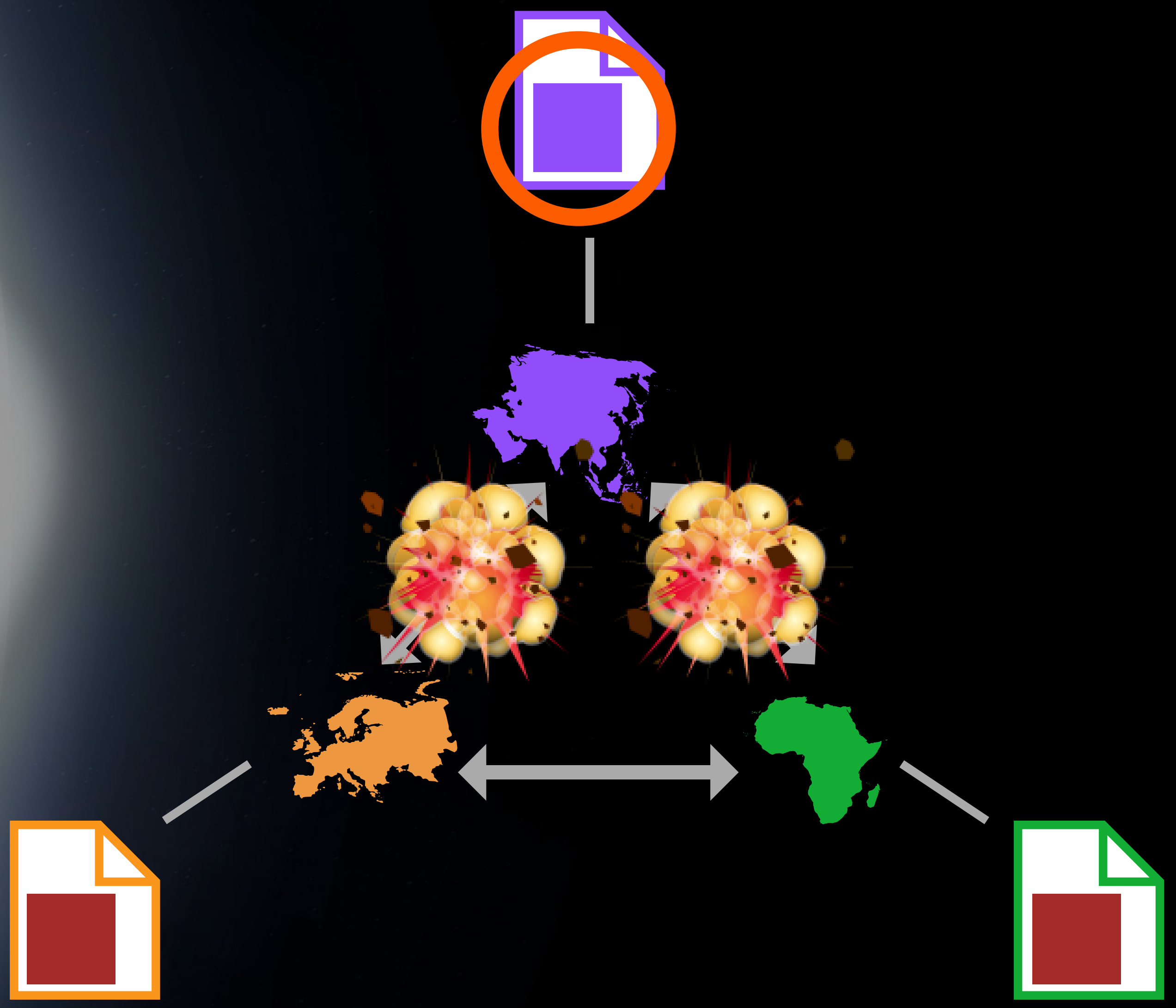
Consistency

Availability



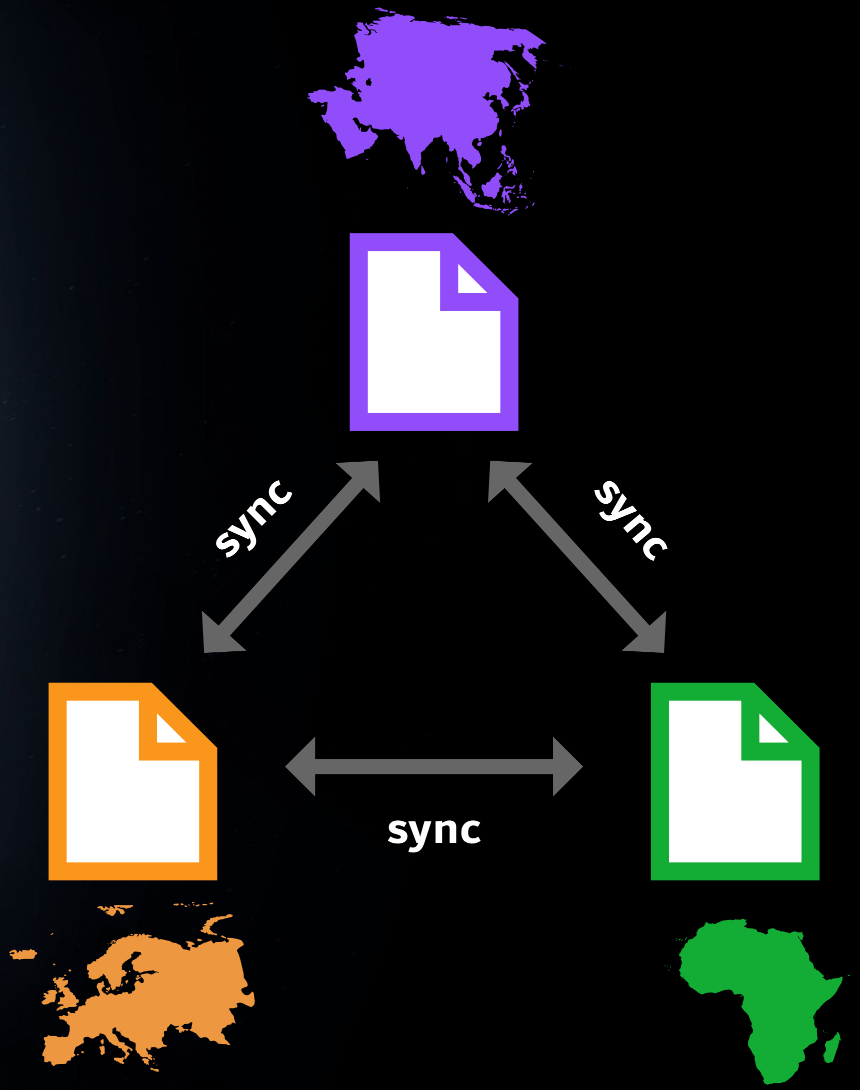
Distributed System: Replicated Data

- ✗ Consistency
- **Availability**



Distributed System: Replicated Data

Eventual Consistency



Conflict-Free Replicated Datatypes (CRDTs)

Conflict-Free Replicated Data Types*

Marc Shapiro[1,5], Nuno Preguiça[1,2], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portugal
[4] UPMC, Paris, France
[5] LIP6, Paris, France

Abstract. Replicating data under Eventual Consistency (EC) allows any replica to accept updates without remote synchronisation. This ensures performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. Under a formal Strong Eventual Consistency (SEC) model, we study sufficient conditions for convergence. A data type that satisfies these conditions is called a Conflict-free Replicated Data Type (CRDT). Replicas of any CRDT are guaranteed to converge in a self-stabilising manner, despite any number of failures. This paper formalises two popular approaches (state- and operation-based) and their relevant sufficient conditions. We study a number of useful CRDTs, such as sets with clean semantics, supporting both *add* and *remove* operations, and consider in depth the more complex Graph data type. CRDT types can be composed to develop large-scale distributed applications, and have interesting theoretical properties.

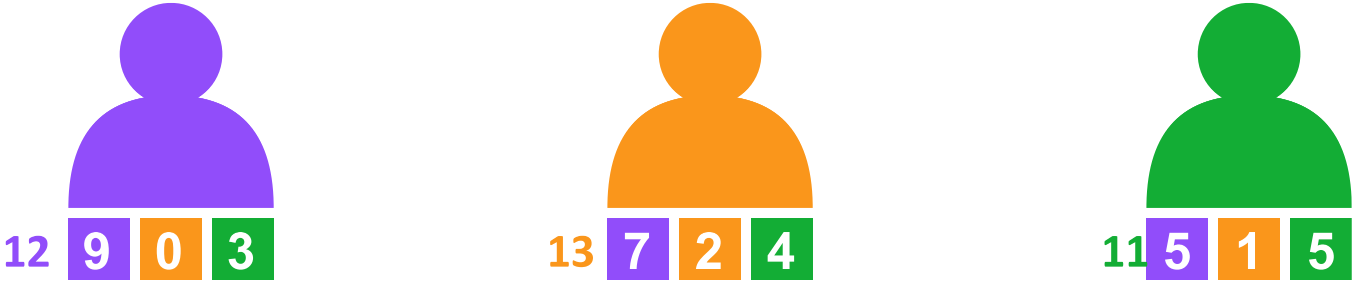
Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems.

Conflict-Free Replicated Datatypes (CRDTs)

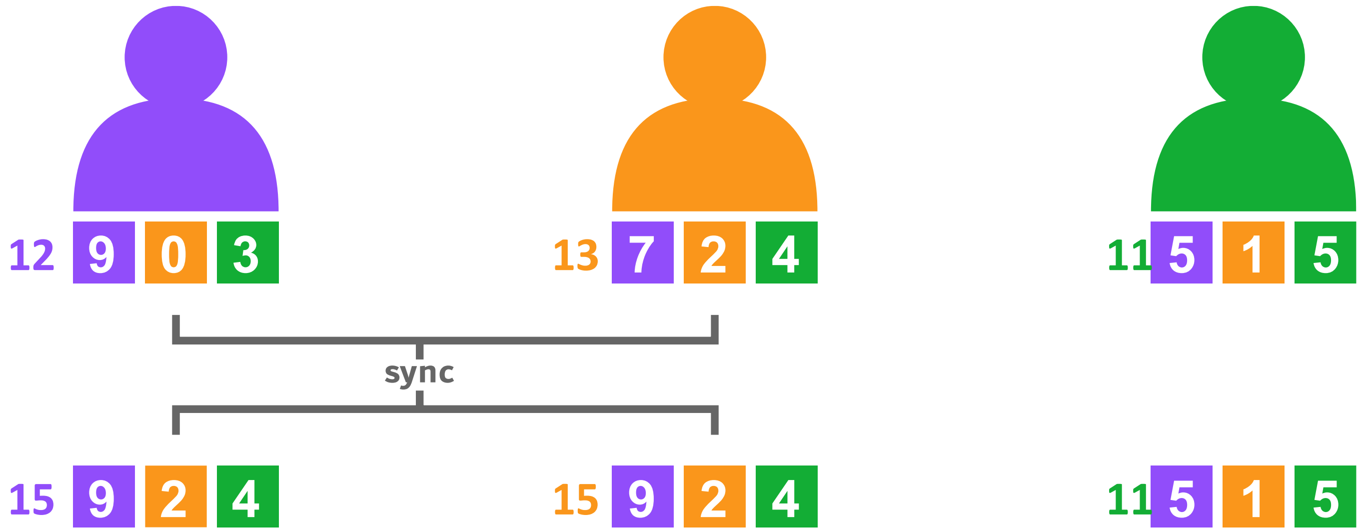
Conflict-Free Replicated Datatypes (CRDTs)

Conflict-Free Replicated Datatypes (CRDTs)

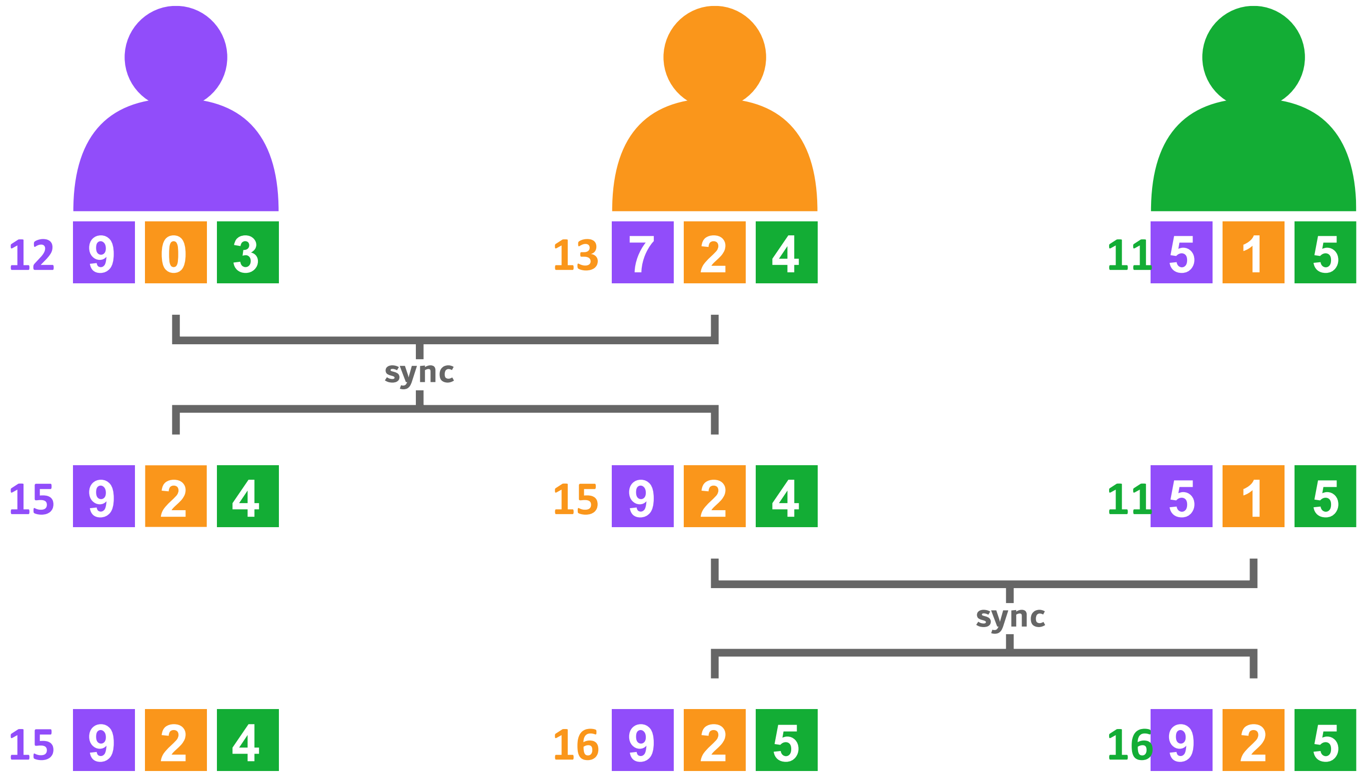
Conflict-Free Replicated Datatypes (CRDTs)



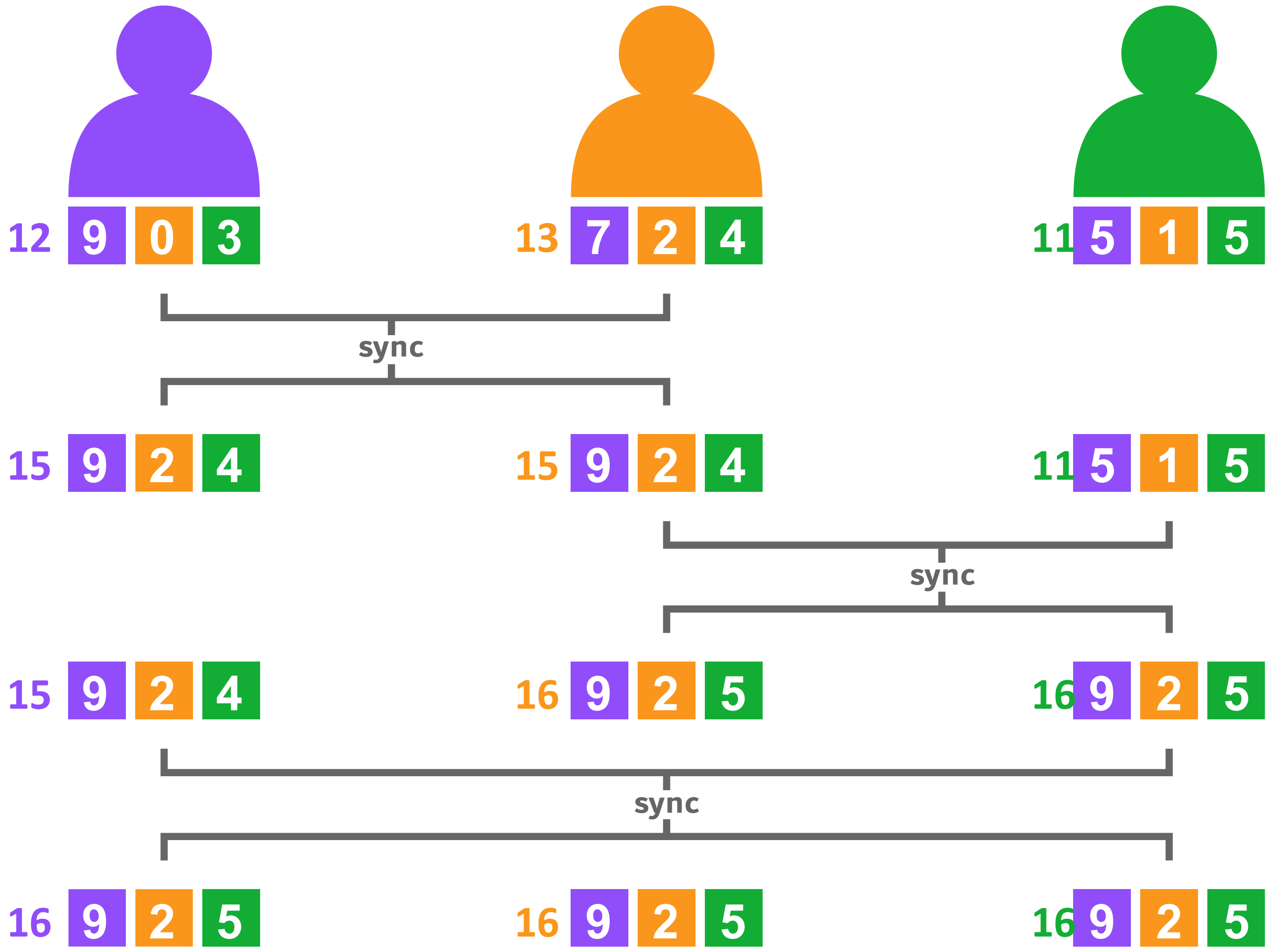
Conflict-Free Replicated Datatypes (CRDTs)



Conflict-Free Replicated Datatypes (CRDTs)



Conflict-Free Replicated Datatypes (CRDTs)



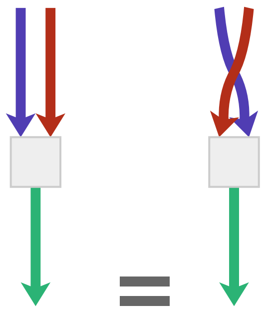
CRDTs Guarantee Eventual Consistency

Eventual Consistency is guaranteed when syncing is:

CRDTs Guarantee Eventual Consistency

Eventual Consistency is guaranteed when syncing is:

Commutative
The order is irrelevant

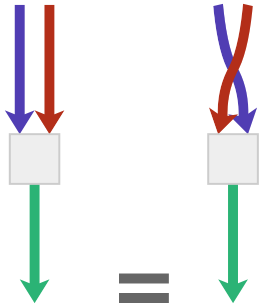


f(x, y) = f(y, x)

CRDTs Guarantee Eventual Consistency

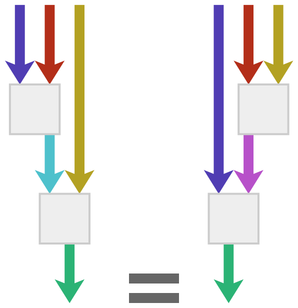
Eventual Consistency is guaranteed when syncing is:

Commutative
The order is irrelevant



$f(x, y) = f(y, x)$

Associative
The grouping is irrelevant

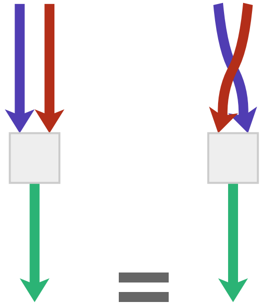


$f(x, f(y, z)) = f(f(x, y), z)$

CRDTs Guarantee Eventual Consistency

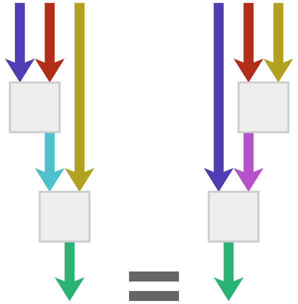
Eventual Consistency is guaranteed when syncing is:

Commutative
The order is irrelevant



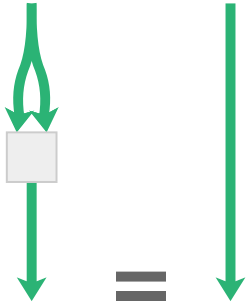
f(x, y) = f(y, x)

Associative
The grouping is irrelevant



f(x, f(y, z)) = f(f(x, y), z)

Idempotent
Syncing equals adds no new information



f(x, x) = x

CRDTs Beyond Simple Counters

Counters: Growing Counter, Positive-Negative Counter, Bounded Counters, …

Sets: Growing Sets, Two-Phase Set, Observed-Remove Set, …

Maps: Add-wins Observed-remove map, …

Registers: Last-Write-Wins, …

Building Correct CRDTs Is Hard

Martin Kleppmann @martin@nondeterministic.computer
@martinkl

Today in "distributed systems are hard": I wrote down a simple CRDT algorithm that I thought was "obviously correct" for a course I'm teaching. Only 10 lines or so long. Found a fatal bug only after spending hours trying to prove the algorithm correct. 😭

11:48 PM · Nov 12, 2020

38 Retweets **5** Quotes **537** Likes **65** Bookmarks

Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode

Martin Kleppmann

Abstract

Designing algorithms for distributed systems has a reputation of being a difficult and error-prone task, but this difficulty is rarely measured or quantified in any way. This report tells the story of one informal experiment, in which users on Twitter were invited to identify the bug in an incorrect CRDT algorithm. Over the following 11 hours, at least 16 people (many of whom are professional software engineers) made attempts to find the bug, but most were unsuccessful. The two people who did identify the bug were both PhD students specialising in CRDTs. This result may serve as evidence of the difficulty of designing correct CRDT algorithms.

Propel

Type-Check CRDT Convergence

Γ

CRDT Type System

Propel

Type-Check CRDT Convergence

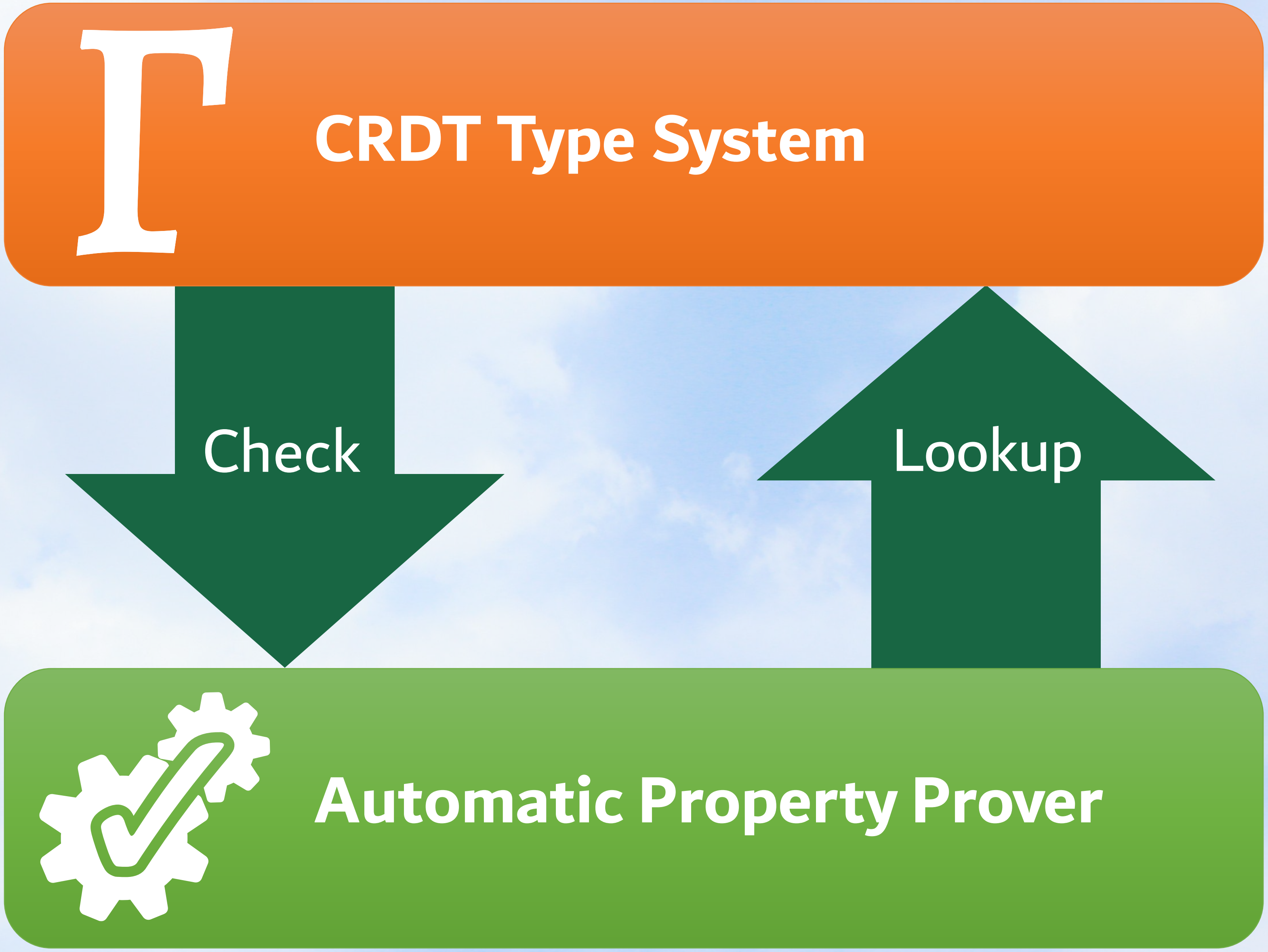
Γ CRDT Type System

Check

Automatic Property Prover

Propel

Type-Check CRDT Convergence



Properties in Types

```
def merge =
    prop.rec[(Comm & Assoc & Idem) := (List[Num], List[Num]) =>: List[Num]]: merge =>
        case (Nil, _) => Nil
        case (_, Nil) => Nil
        case (x :: xs, y :: ys) => max(y, x) :: merge(xs, ys)
```

Properties in Types

```
def merge =
    prop.rec[(Comm & Assoc & Idem) := (List[Num], List[Num]) =>: List[Num]]: merge =>
        case (Nil, _) => Nil
        case (_, Nil) => Nil
        case (x :: xs, y :: ys) => max(y, x) :: merge(xs, ys)
```

Properties in Types

```
def merge =
    prop.rec[(Comm & Assoc & Idem) := (List[Num], List[Num]) =>: List[Num]]: merge =>
        case (Nil, _) => Nil
        case (_, Nil) => Nil
        case (x :: xs, y :: ys) => max(y, x) :: merge(xs, ys)
```

Properties in Types

```
def merge =
    prop.rec[(Comm & Assoc & Idem) := (List[Num], List[Num]) =>: List[Num]]: merge =>
        case (Nil, _) => Nil
        case (_, Nil) => Nil
        case (x :: xs, y :: ys) => max(y, x) :: merge(xs, ys)


def max =
    prop.rec[(Comm & Assoc & Idem) := (Num, Num) =>: Num]: max =>
        ...
```

Properties in Types

```
def merge =
    prop.rec[(Comm & Assoc & Idem) := (List[Num], List[Num]) =>: List[Num]]: merge =>
        case (Nil, _) => Nil
        case (_, Nil) => Nil
        case (x :: xs, y :: ys) => max(y, x) :: merge(xs, ys)


def max =
    prop.rec[(Comm & Assoc & Idem) := (Num, Num) =>: Num]: max =>
        ...
```

Properties in Types Enable Composition

```
def zipWith[P >: (Comm & Assoc & Idem), T] =
    prop.rec[(P := (T, T) =>: T) => (P := (List[T], List[T]) =>: List[T])]:
        zipWith => f =>
            case (Nil, y) => y
            case (x, Nil) => x
            case (x :: xs, y :: ys) => f(x, y) :: zipWith(f)(xs, ys)
```

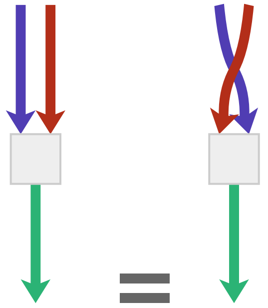
Properties in Types Enable Composition

```
def zipWith[P >: (Comm & Assoc & Idem), T] =
    prop.rec[(P := (T, T) =>: T) => (P := (List[T], List[T]) =>: List[T])]:
        zipWith => f =>
            case (Nil, y) => y
            case (x, Nil) => x
            case (x :: xs, y :: ys) => f(x, y) :: zipWith(f)(xs, ys)


def merge =
    prop[(Comm & Assoc & Idem) := (List[Num], List[Num]) =>: List[Num])]:
        zipWith(max)
```

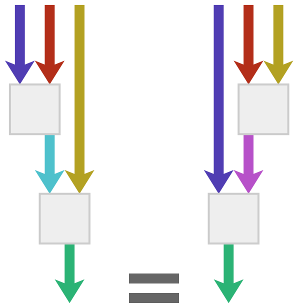
Properties of Binary Functions

Commutativity
The sync order is irrelevant



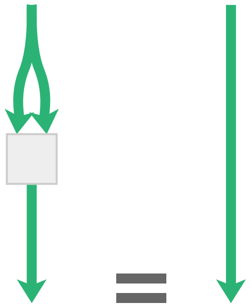
f(x, y) = f(y, x)

Associativity
The sync order is still irrelevant



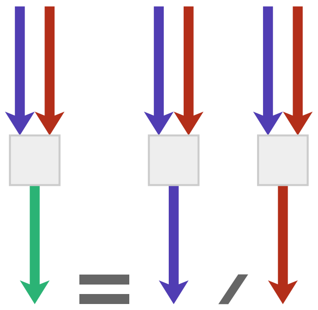
f(x, f(y, z)) = f(f(x, y), z)

Idempotency
Syncing equals adds no new information



f(x, x) = x

Selectivity
A function will always return one of its argument.

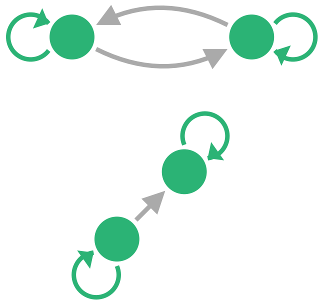


f(x, y) = x ∨ f(x, y) = y

Example: projections, maximum, conditionals

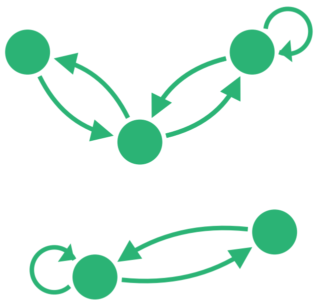
Properties of Binary Relations

Reflexivity



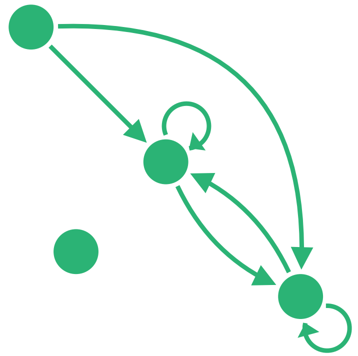
$f(x, x) = \top$

Symmetry



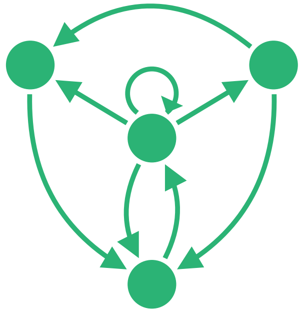
$f(x, y) = \top$
$\vdash f(y, x) = \top$

Transitivity



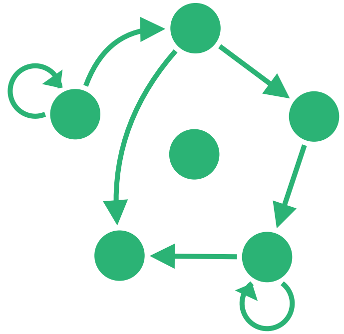
$f(x, y) = \top \wedge f(y, z) = \top$
$\vdash f(x, z) = \top$

Connectivity



$f(x, y) = \top \vee f(y, x) = \top$

Antisymmetry



$f(x, y) = \top \wedge x \neq y$
$\vdash f(y, x) = \bot$

Propel Proofs Example

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs Example

```
max(a, b) = max(b, a)
```

Goal

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs Example

Goal

```
max(a, b) = max(b, a)
```

Typing Context

```
Γ =
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs Example

Goal

```
max(a, b) = max(b, a)
```

Typing Context

```
Γ =
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec
```

Equalities

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs Example

Goal

```
max(a, b) = max(b, a)
```

Typing Context

```
Γ =
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec
```

Equalities

Inequalities

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Case Analysis

Goal

```
max(Nil, b) = max(b, Nil)
```

Typing Context

```
Γ = max:      Comm                       := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec
```

Equalities

```
a = Nil
```

Inequalities

```
a ≠ B0(a')
a ≠ B1(a')
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Reflexivity

Goal

```
b = b
```

Typing Context

```
Γ = max:     Comm                         := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

Inequalities

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Next Case

Goal

```
B0(max(x, y)) = B0(max(y, x))
```

Typing Context

```
Γ = max:     Comm                       := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B0(y)
```

Inequalities

```
a ≠ Nil
a ≠ B1(a')
b ≠ Nil
b ≠ B1(b')
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Constructor Elimination

Goal

```
B0(max(x, y)) = B0(max(y, x))
```

Typing Context

```
Γ = max:     Comm                       := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans)     := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B0(y)
```

Inequalities

```
a ≠ Nil        b ≠ B1(b')
a ≠ B1(a')
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Constructor Elimination

Goal

```
max(x, y) = max(y, x)
```

Typing Context

```
Γ = max:     Comm                       := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B0(y)
```

Inequalities

```
a ≠ Nil          b ≠ B1(b')
a ≠ B1(a')
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Use Properties from Γ

Goal

```
max(x, y) = max(y, x)
```

Typing Context

```
Γ  max:     Comm                        := (BitVec, BitVec) =>: BitVec,
   equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
   a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B0(y)
max(x, y) = max(y, x)
```

Inequalities

```
a ≠ Nil          b ≠ B1(b')
a ≠ B1(a')
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Use a Known Equality

Goal

```
max(y, x) = max(y, x)
```

Typing Context

```
Γ = max:     Comm                      := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B0(y)
max(x, y) = max(y, x)
```

Inequalities

```
a ≠ Nil          b ≠ B1(b')
a ≠ B1(a')
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Next Case

Goal

```
if equals(max(x,y),y) then B1(y) else B0(x) = if equals(max(y,x),x) then B1(y) else B0(x)
```

Typing Context

```
Γ = max:     Comm                          := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B1(y)
```

Inequalities

```
a ≠ Nil          b ≠ B0(b')
a ≠ B1(a')
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: "if" is a case expression

Goal

```
B1(y) = if equals(max(y,x),x) then B1(y) else B0(x)
```

Typing Context

```
Γ = max:      Comm                        := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B1(y)
            equals(max(x,y),y)  = T
```

Inequalities

```
a ≠ Nil          b ≠ B0(b')
a ≠ B1(a')
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: "if" is a case expression

Goal

```
B1(y) = if equals(max(y,x),x) then B1(y) else B0(x)
```

Typing Context

```
Γ = max:     Comm                        := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B1(y)
max(x,y) = y   equals(max(x,y),y) = T
```

Inequalities

```
a ≠ Nil          b ≠ B0(b')
a ≠ B1(a')
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Next Case

Goal

```
B1(y) = B0(x)
```

Typing Context

```
Γ = max:     Comm                         := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)        max(x, y) = x
b = B1(y)        equals(max(x,y),x)  = ⊥
max(x,y) = y     equals(max(x,y),y)  = T
```

Inequalities

```
a ≠ Nil          b ≠ Bit0(b')
a ≠ Bit1(a')
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Stuck

Goal

```
B1(y) = B0(x)
```

Typing Context

```
Γ = max:     Comm                       := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B1(y)        equals(max(x,y),x)  = ⊥
max(x,y) = y     equals(max(x,y),y)  = ⊤
```

Inequalities

```
a ≠ Nil          b ≠ B0(b')
a ≠ B1(a')       max(y, x) ≠ x
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Counterexample

The case `a=B0(x)` and `b=B1(y)` may not be commutative.

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Counterexample

The case `a=B0(x)` and `b=B1(y)` may not be commutative.

Manual inspection: `max(B1(B0(Z)), B0(Z)) != max(B0(Z), B1(B0(Z)))`

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
```

Propel Proofs: Counterexample

The case a=B0(x) and b=B1(y) may not be commutative.

Manual inspection: max(B1(B0(Z)), B0(Z)) != max(B0(Z), B1(B0(Z)))

```
--- case (B1(x), B0(y)) => if equals(max(x, y), y) then B1(x) else B0(y)
+++ case (B1(x), B0(y)) => if equals(max(x, y), x) then B1(x) else B0(y)
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), x) then B1(x) else B0(y)
```

Propel Proofs: Contradiction

Goal

```
B1(y) = B0(y)
```

Typing Context

```
Γ = max:      Comm                          := (BitVec, BitVec) =>: BitVec,
    equals: (Antisym & Sym & Trans) := (BitVec, BitVec) =>: Bool,
    a: BitVec, b: BitVec, a': BitVec, b': BitVec, x: BitVec, y: BitVec
```

Equalities

```
a = B0(x)
b = B1(y)          equals(max(x,y),y) = ⊥
max(x,y) = y       equals(max(x,y),y) = ⊤
```

Inequalities

```
a ≠ Nil            b ≠ B0(b')
a ≠ B1(a')         max(x, y) ≠ y
b ≠ Nil
```

```
def max = prop.rec[Comm := (BitVec, BitVec) =>: BitVec]: max =>
  (a, b) => (a, b) match
    case (Nil, b) => b
    case (a, Nil) => a
    case (B0(x), B0(y)) => B0(max(x, y))
    case (B1(x), B1(y)) => B1(max(x, y))
    case (B0(x), B1(y)) => if equals(max(x, y), y) then B1(y) else B0(x)
    case (B1(x), B0(y)) => if equals(max(x, y), x) then B1(x) else B0(y)
```

Equality Lifting: equals(x, y) = T ⊢ x = y

`equals(max(x,y), y) = T` → Is it okay? → `max(x,y) = y`

Equality Lifting: equals(x, y) = T ⊢ x = y

`equals(max(x,y), y) = T` `max(x,y) = y`

Is it okay?

Theorem: Equality is the only antisymmetric, symmetric, and transitive relation

Equality Lifting: equals(x, y) = T ⊢ x = y

`equals(max(x,y), y) = T` → `max(x,y) = y`

Is it okay?

Theorem: `equals:` **`Antisym & Sym & Trans`** `:= (A,A) =>: Bool, equals(x, y) = T ⊢ x = y`

Theorem: `equals:` **`Antisym & Sym & Trans`** `:= (A,A) =>: Bool, equals(x, y) = ⊥ ⊢ x ≠ y`

Equality Lifting: equals(x, y) = T ⊢ x = y

`equals(max(x,y), y) = T` → `max(x,y) = y`

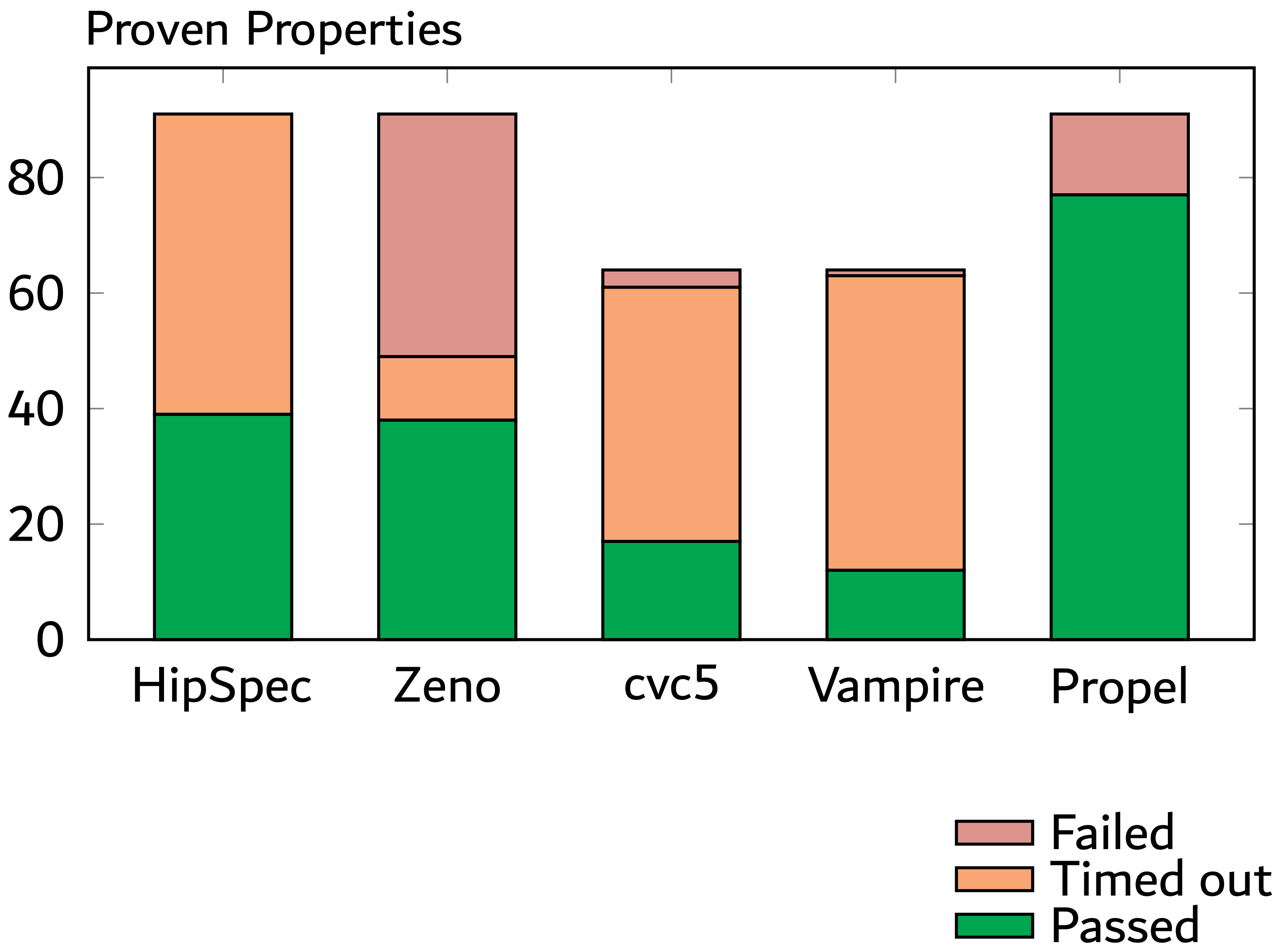
Is it okay?

Propel discovers these theorems without any ad-hoc rules

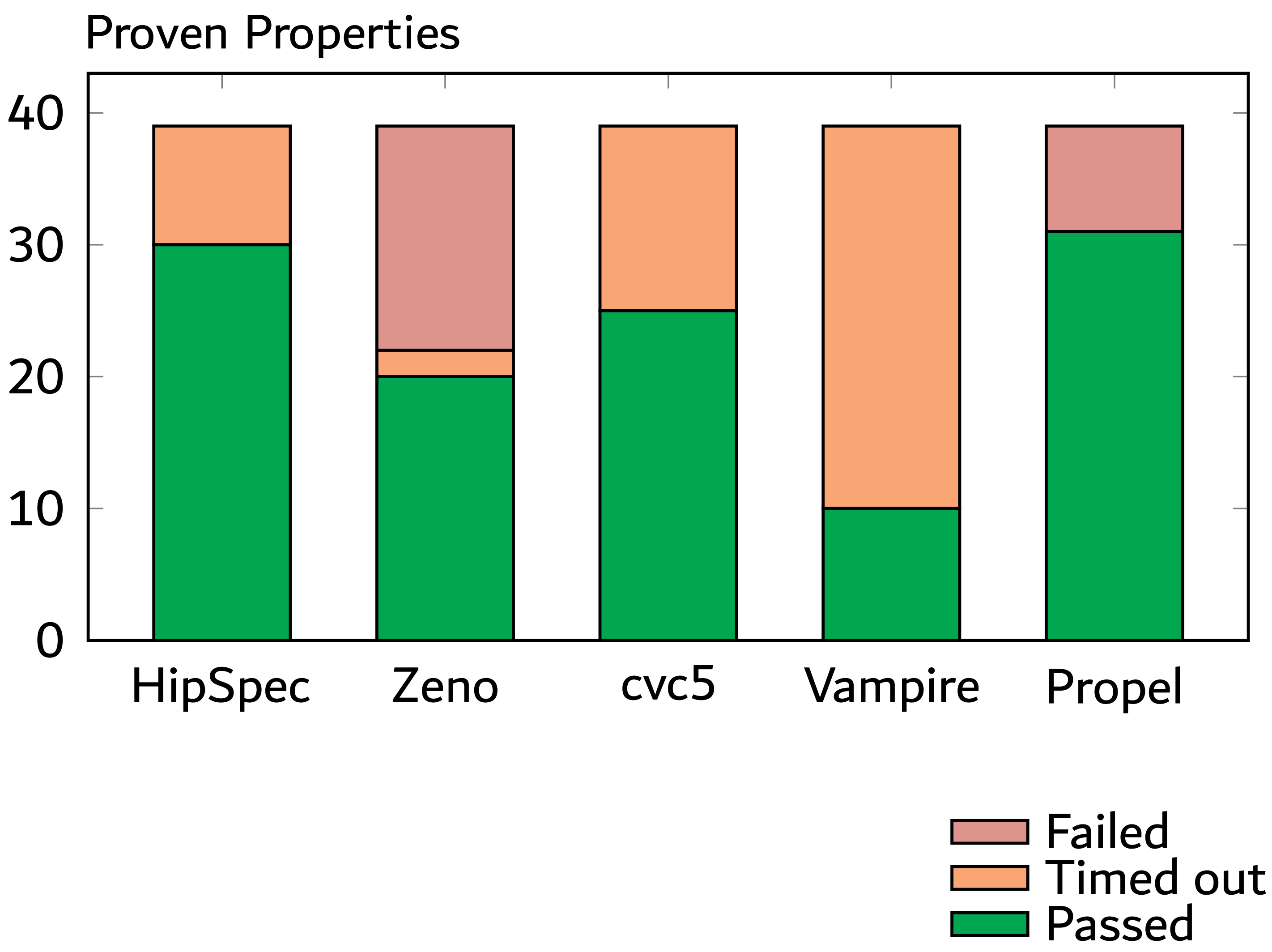
Theorem: `equals: Antisym & Sym & Trans := (A,A) =>: Bool, equals(x, y) = T ⊢ x = y`

Theorem: `equals: Antisym & Sym & Trans := (A,A) =>: Bool, equals(x, y) = ⊥ ⊢ x ≠ y`

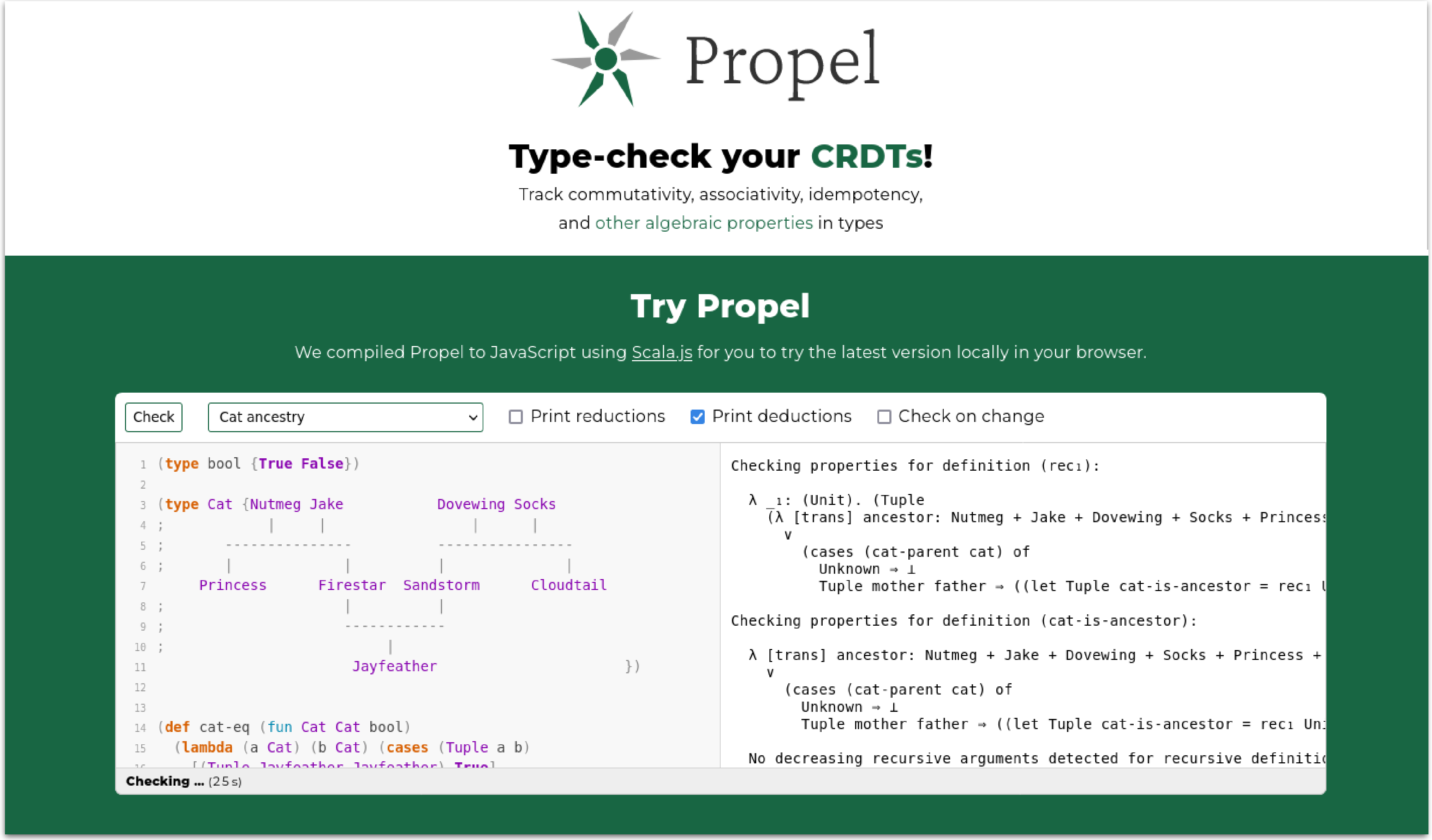
Propel for CRDTs



Propel Beyond CRDTs



propel-prover.github.io



propel-prover.github.io