An introduction to verification of replicated data types

Dr. Kevin De Porre

Who am I?

- Dr. Kevin De Porre
- Obtained my PhD last year at VUB
  - Research focused on design, implementation and verification of RDTs
- Currently:
  - Founding engineer @ ElectricSQL
    - Proud sponsor of DARE
  - Part-time postdoctoral researcher @ VUB

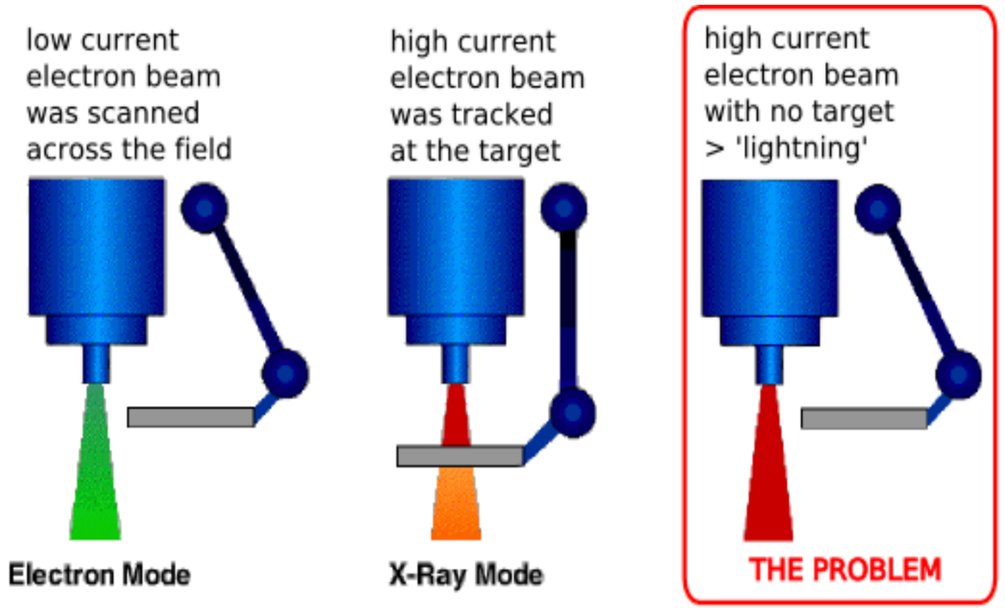
What this lecture is about

- You already learned about Replicated Data Types (RDTs)
  - e.g. CRDTs

- This lecture is about formal verification of RDTs and covers:
  - Several verification strategies
  - Automated verification with SMT solving
  - VeriFx: a verification language atop SMT solvers
  - Applied verification of CRDTs using VeriFx

Why we need formal verification

- To ensure correctness, i.e., avoid bugs!
- Bugs can have disastrous consequences



tray including the target, a flattening filter, the collimator jaws and an ion chamber was moved OUT for "electron" mode, and IN for "photon" mode.

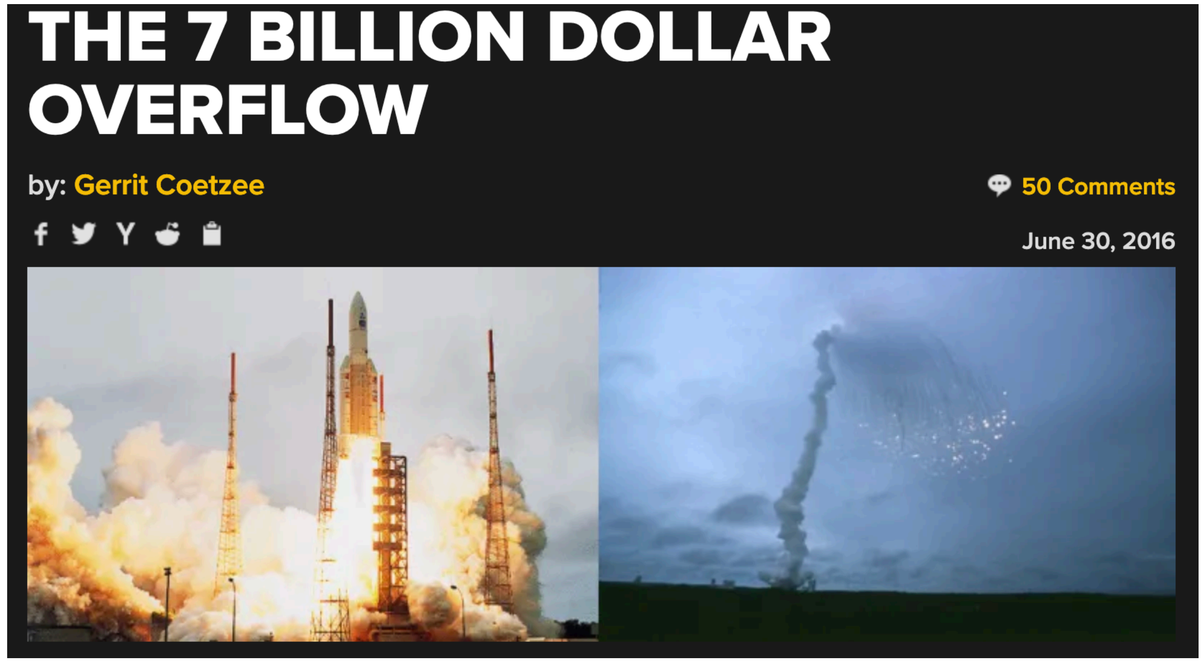
Source: https://www.cs.princeton.edu/courses/archive/spr16/cos217/lectures/24_ProgramVerif.pdf

Therac-25 radiation therapy machine to treat cancer.

Critical bug caused 6 deaths between 1985 and 1987!

Why we need formal verification

- To ensure correctness, i.e., avoid bugs!
- Bugs can have disastrous consequences



Source: https://hackaday.com/2016/06/30/fail-of-the-week-in-1996-the-7-billion-dollar-overflow/

Ariane V rocket (1996)

Overflow due to typecast from 64-bit floating point to 16-bit signed integer!

A note on software testing

- Widespread technique to unravel bugs
- Ideally we test all paths through the program
  - Not feasible for complex programs —> combinatorial explosion
  - Instead, we test frequent cases and *known* corner cases
- *Unknown* corner cases are the problem!
  - Remain untested…
  - Especially problematic for RDTs
    —> many corner cases due to ordering of concurrent operations

A note on software testing

- *Unknown* corner cases are the problem!
  - Remain untested…
  - Especially problematic for RDTs
    —> many corner cases due to ordering of concurrent operations

- Complexity of RDTs showcased by bugs in Operational Transformation (OT)
  - > 15 years of research, > 6 internationally peer-reviewed papers
  - All shown to be wrong due to corner cases regarding the ordering of operations!
  - cf. Imine et al. (ECSCW'03) for more information 

A note on software testing

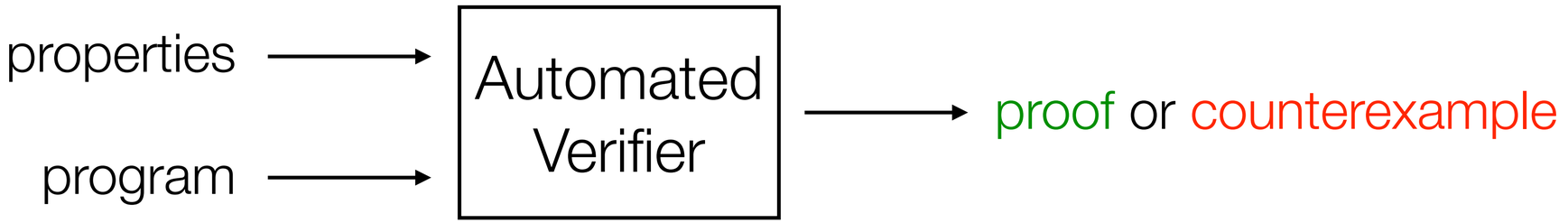
- *Unknown* corner cases are the problem!
  - Remain untested…
  - Especially problematic for RDTs
    —> many corner cases due to ordering of concurrent operations

- Complexity of RDTs showcased by bugs in Operational Transformation (OT)
  - > 15 years of research, > 6 internationally peer-reviewed papers
  - All shown to be wrong due to corner cases regarding the ordering of operations!
  - cf. Imine et al. (ECSCW'03) for more information 

> "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." — Edsger W. Dijkstra

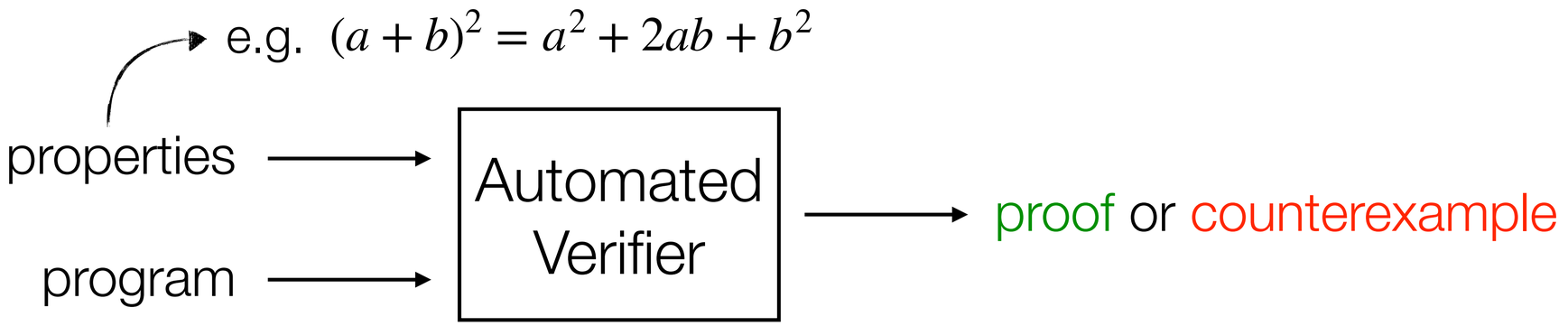
Software verification

- Aims to prove that a program is correct
- Requires a specification of the correctness properties to verify
  - depends on the application at hand, i.e. is application-specific
- Ideally:



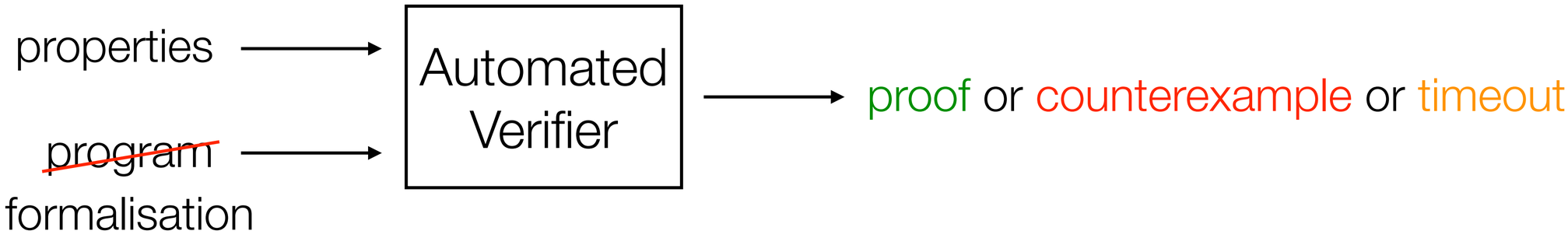
Software verification

- Aims to prove that a program is correct
- Requires a specification of the correctness properties to verify
  - depends on the application at hand, i.e. is application-specific
- Ideally:



Software verification

- Aims to prove that a program is correct
- Requires a specification of the correctness properties to verify
  - depends on the application at hand, i.e. is application-specific
- Often:



Software verification strategies

- Paper proofs
  - Use logic to reason about program and prove correctness on paper
  - Aim to convince the reader that the reasoning is correct and the proof holds
  - ✓ High familiarity —> taught at school in math classes
  - x Logic is unchecked —> reasoning flaws render the entire proof obsolete
- Mechanised proofs
  - ✓ Proof logic is *machine-checked* —> more convincing
  - x Use advanced tools and languages that require much more expertise

Types of mechanical verification

- Three types of mechanical verification:
  - Interactive verification tools
    - Generate verification conditions (VCs)
    - User input provided *after* VC generation to prove correctness
  - Auto-active verification tools
    - User input provided *before* VC generation
    - User input acts as hints to help the tool verify the VCs automatically
  - Automated verification tools
    - No additional user input required besides the program and its correctness properties
    - Generate the VCs and verify them automatically

Types of mechanical verification

- Three types of mechanical verification:
  - Interactive verification tools
    - Generate verification conditions (VCs)
    - User input provided *after* VC generation to prove correctness
  - Auto-active verification tools
    - User input provided *before* VC generation
    - User input acts as hints to help the tool verify the VCs automatically
  - Automated verification tools
    - No additional user input required besides the program and its correctness properties
    - Generate the VCs and verify them automatically

Interactive verification

- Includes interactive theorem provers (aka proof assistants)
  - Language exposes proof tactics
  - Tactics are logic reasoning steps that can be used in proofs
  - e.g., Coq and Isabelle

Interactive verification

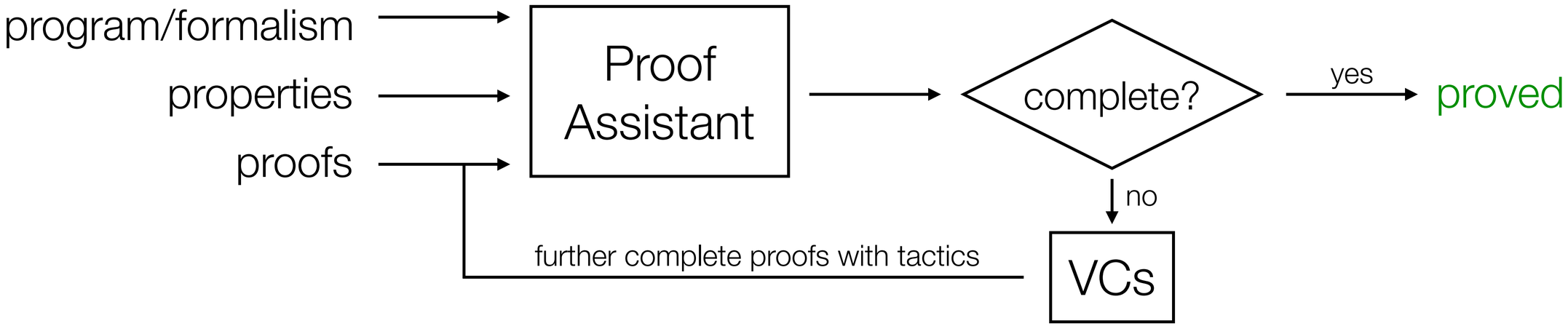
- Includes interactive theorem provers (aka proof assistants)
  - Language exposes proof tactics
  - Tactics are logic reasoning steps that can be used in proofs
  - e.g., Coq and Isabelle
- Need to implement the program or formalism in the prover’s language

Interactive verification

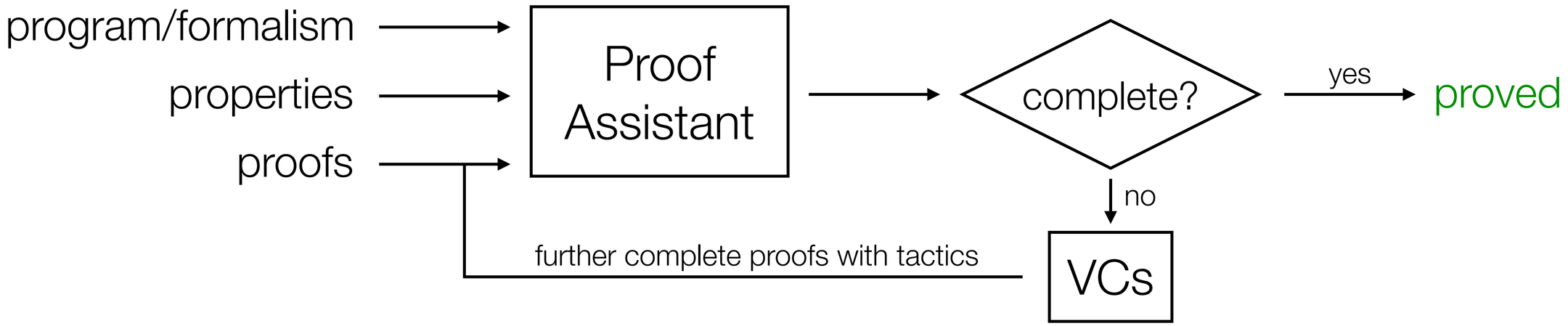
- Includes interactive theorem provers (aka proof assistants)
  - Language exposes proof tactics
  - Tactics are logic reasoning steps that can be used in proofs
  - e.g., Coq and Isabelle
- Need to implement the program or formalism in the prover’s language
- Manually define and verify correctness properties using proof tactics

Interactive verification

- Includes interactive theorem provers (aka proof assistants)
  - Language exposes proof tactics
  - Tactics are logic reasoning steps that can be used in proofs
  - e.g., Coq and Isabelle
- Need to implement the program or formalism in the prover's language
- Manually define and verify correctness properties using proof tactics

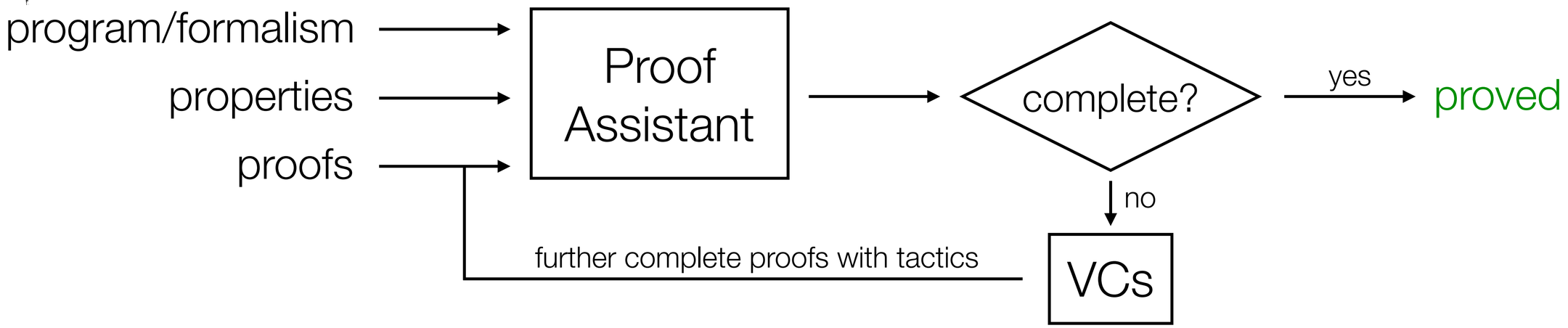


Interactive verification



Interactive verification

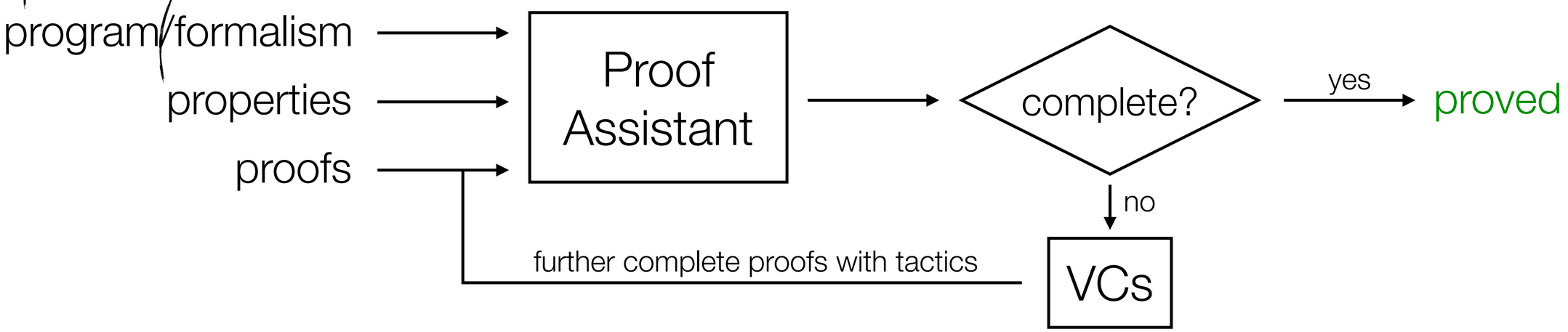
```
Inductive seq : nat -> Set :=
| niln : seq 0
| consn : forall n : nat, nat -> seq n -> seq (S n).

Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
  match s with
  | niln => 0
  | consn i _ s' => S (length i s')
  end.
```



Interactive verification

```
Inductive seq : nat -> Set :=
| niln : seq 0
| consn : forall n : nat, nat -> seq n -> seq (S n).

Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
  match s with
  | niln => 0
  | consn i _ s' => S (length i s')
  end.
```

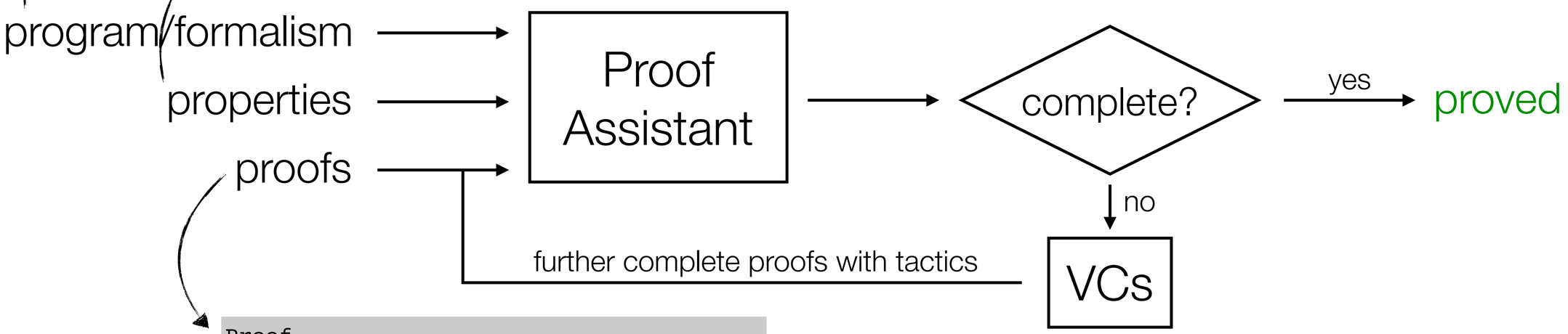
```
Theorem length_corr : forall (n : nat) (s : seq n), length n s = n.
```



Interactive verification

```
Inductive seq : nat -> Set :=
| niln : seq 0
| consn : forall n : nat, nat -> seq n -> seq (S n).

Fixpoint length (n : nat) (s : seq n) {struct s} : nat :=
  match s with
  | niln => 0
  | consn i _ s' => S (length i s')
  end.
```

```
Theorem length_corr : forall (n : nat) (s : seq n), length n s = n.
```

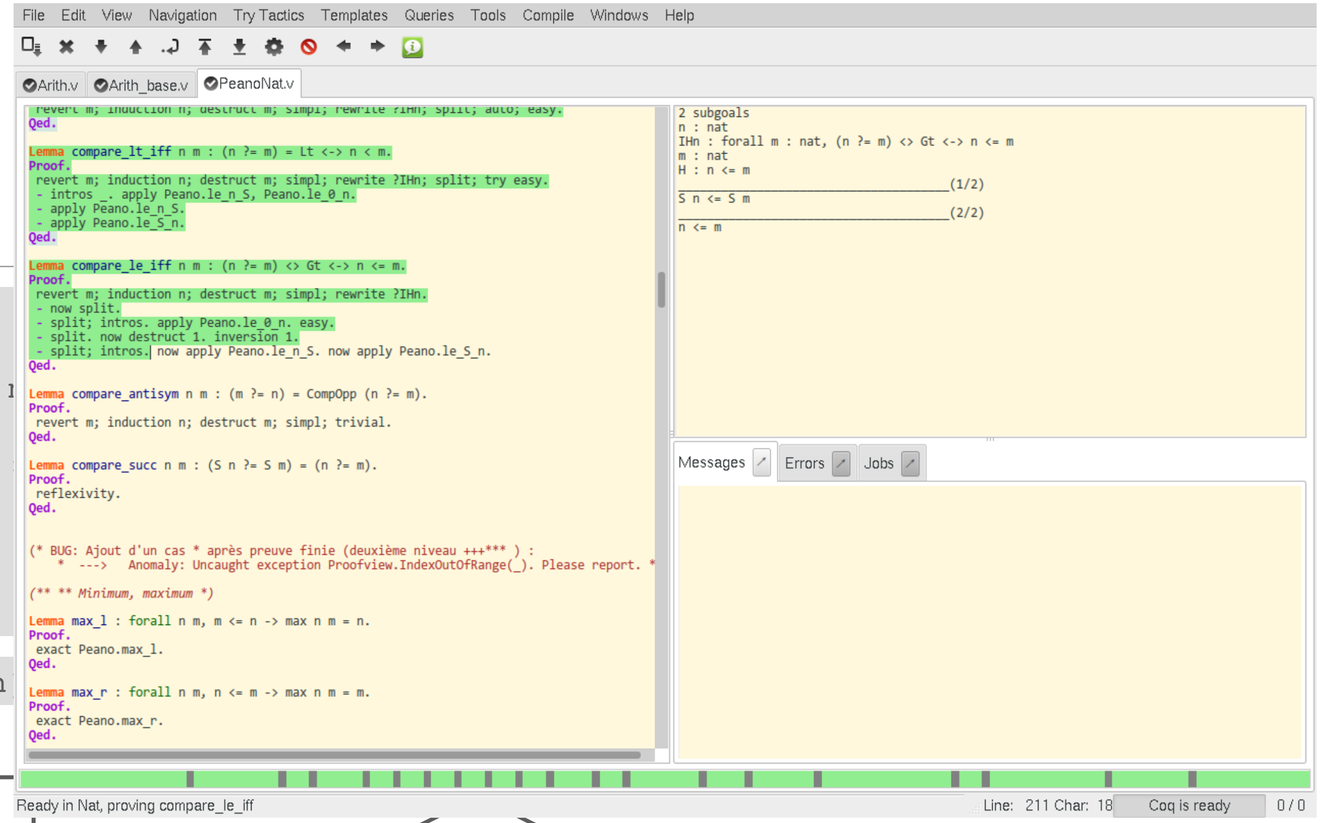



```
Proof.
  intros n s. induction s. simpl.
  trivial. simpl. rewrite IHs. trivial.
Qed.
```

Interactive verification

```
Inductive seq : nat -> Set :=
| niln : seq 0
| consn : forall n : nat, nat -> seq n -> seq (S n

Fixpoint length (n : nat) (s : seq n) {struct s}
  match s with
  | niln => 0
  | consn i _ s' => S (length i s')
  end.
```

```
Theorem length_corr : forall (n : nat) (s : seq n
```



program/formalism

properties

proofs

Proof Assistant

complete?

yes

proved

no

VCs

further complete proofs with tactics

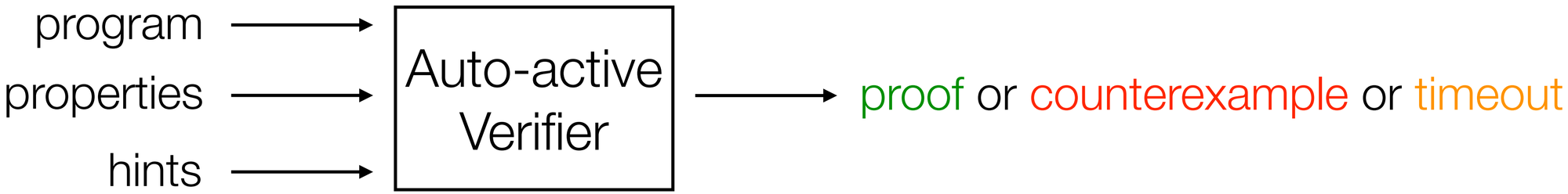
```
Proof.
  intros n s. induction s. simpl.
  trivial. simpl. rewrite IHs. trivial.
Qed.
```

Types of mechanical verification

- Three types of mechanical verification:
  - Interactive verification tools
    - Generate verification conditions (VCs)
    - User input provided *after* VC generation to prove correctness
  - Auto-active verification tools
    - User input provided *before* VC generation
    - User input acts as hints to help the tool verify the VCs automatically
  - Automated verification tools
    - No additional user input required besides the program and its correctness properties
    - Generate the VCs and verify them automatically

Auto-active verification

- Automatic but programs must be extended with additional hints:
  - method preconditions and postconditions
  - decrease annotations to check termination
  - loop invariants
  - assertions
- Languages like Dafny, Boogie, VCC, etc.



Auto-active verification

- Verified Fibonacci program in Dafny

```
function fib(n: nat): nat
{
  if n == 0 then 0
  else if n == 1 then 1
  else fib(n - 1) + fib(n - 2)
}

method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n)
{
  if n == 0 { return 0; }
  var i: int := 1;
  var a := 0;
  b := 1;
  while i < n
    invariant 0 < i <= n
    invariant a == fib(i - 1)
    invariant b == fib(i)
  {
    a, b := b, a + b;
    i := i + 1;
  }
}
```

correctness property

- expressed as a postcondition

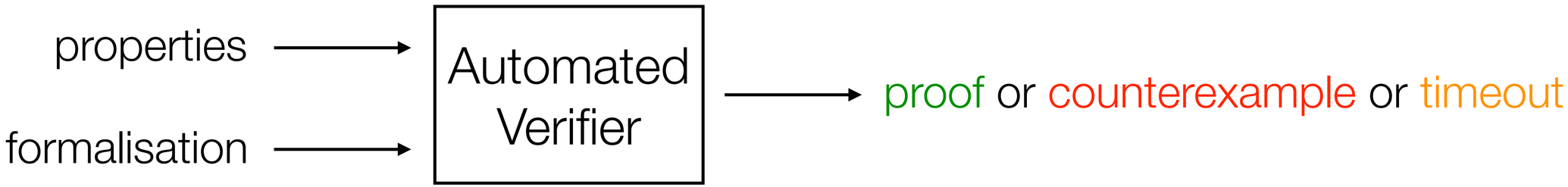
loop invariants

Types of mechanical verification

- Three types of mechanical verification:
  - Interactive verification tools
    - Generate verification conditions (VCs)
    - User input provided *after* VC generation to prove correctness
  - Auto-active verification tools
    - User input provided *before* VC generation
    - User input acts as hints to help the tool verify the VCs automatically
  - Automated verification tools
    - No additional user input required besides the program and its correctness properties
    - Generate the VCs and verify them automatically

Automatic verification

- Fully automated, require no additional hints
- Promising but… no automated verifiers for mainstream languages
- Instead, they verify formal specifications in logic theories
- Examples: model checkers (TLA+, etc.), SMT solvers (Z3, CVC4, etc.)



SMT Solving

- Satisfiability Modulo Theories (SMT)

- Solvers determine satisfiability of first-order logic formulas
  - by searching for a model that satisfies the formula
  - e.g., `a: Int, check(a * a = a)`
    - outcome: satisfiable, model: `a = 0`
    - this proves $\exists a : a * a = a$ but not $\forall a : a * a = a$

- How can we leverage SMT solvers to prove a formula φ ?

SMT Solving

- Satisfiability Modulo Theories (SMT)

- Solvers determine satisfiability of first-order logic formulas
  - by searching for a model that satisfies the formula
  - e.g., `a: Int, check(a * a = a)`
    - outcome: satisfiable, model: `a = 0`
    - this proves $\exists a : a * a = a$ but not $\forall a : a * a = a$

- How can we leverage SMT solvers to prove a formula φ ?
  - by checking that the negation is unsatisfiable, i.e., `check(` $\neg\varphi$ `) == UNSAT`
  - $\neg\varphi$ is a counterexample for φ
  - If $\neg\varphi$ is unsatisfiable then no counter example exists, i.e. φ is always true

Expressiveness of SMT Solvers

- There are SMT theories for:
  - data types, arrays, floating point, bit vectors, strings, etc.
  - different solvers support different theories
- Some theories or fragments of theories are decidable
  - e.g., Combinatory Array Logic (CAL) is decidable
  - but introduction of quantifiers (∀, ∃) makes it undecidable
- In practice SMT solvers can often handle formulas using undecidable theories
- However, recursive algorithms are problematic
  - Require proofs by induction, not supported by SMT solvers out-of-the-box

Combinatory Array Logic (CAL)

- Array is a mapping from input(s) to output
  - an array of type `(Array A B)` maps values of type A to values of type B
  - arrays are defined for all inputs!
    - much like functions

- Z3 treats arrays as function spaces
  - ➡ functions can be turned into arrays using lambdas
  - e.g., `(lambda ((x Int)) (+ x 1))` defines an array mapping integers to their successor

- Arrays can be multidimensional, e.g. `(Array A B C)`
  - Think of it as a function that takes 2 arguments
  - e.g., `(lambda ((x Int) (y Int)) (+ x y))`

Combinatory Array Logic (CAL)

- Array theory is extensional
  - arrays are considered equal if they return the same output for *all* input
  - $a = b \iff \forall i : a[i] = b[i]$
- We can build common data types on top of arrays
- Let's implement a set data type: `(Set A)`
  - with 5 operations:
    - `new :: () -> (Set A)`
    - `add :: (Set A) -> (Set A)`
    - `remove :: (Set A) -> (Set A)`
    - `contains :: (Set A) -> Bool`
    - `map :: ((A -> B) (Set A)) -> (Set B)`

A Set Data Type in SMT

Set data type implementation:

```
(define-sort Set (A) (Array A Bool))

(define-fun new () (Set Int)
  (lambda ((x Int)) false))

(define-fun add ((s (Set Int)) (x Int)) (Set Int)
  (store s x true))

(define-fun remove ((s (Set Int)) (x Int)) (Set Int)
  (store s x false))

(define-fun contains ((s (Set Int)) (x Int)) Bool
  (select s x))

(define-fun map ((f (Array Int Int)) (s (Set Int))) (Set Int)
  (lambda ((x Int))
    (exists ((y Int))
      (and
        (contains s y)
        (= (select f y) x)))))
```

A Set Data Type in SMT

Set data type implementation:

```
(define-sort Set (A) (Array A Bool))

(define-fun new () (Set Int)
  (lambda ((x Int)) false))

(define-fun add ((s (Set Int)) (x Int)) (Set Int)
  (store s x true))

(define-fun remove ((s (Set Int)) (x Int)) (Set Int)
  (store s x false))

(define-fun contains ((s (Set Int)) (x Int)) Bool
  (select s x))

(define-fun map ((f (Array Int Int)) (s (Set Int))) (Set Int)
  (lambda ((x Int))
    (exists ((y Int))
      (and
        (contains s y)
        (= (select f y) x)))))
```

Let's verify our `map` function:

```
(declare-const s1 (Set Int)) ; some initial set
(declare-const s2 (Set Int)) ; mapped set
(declare-const s3 (Set Int)) ; verification set

; s2 = s1.map(x => x+1)
(assert
  (= s2
     (map (lambda ((x Int)) (+ x 1)) s1)))

; s3 = expected outcome
(assert
  (forall ((x Int))
    (and
      (=> (contains s1 x) (contains s3 (+ x 1)))
      (=> (not (contains s1 x)) (not (contains s3 (+ x 1)))))))

(assert (not (= s2 s3)))

(check-sat) ; outputs unsat -> no counterexamples exist
```

A Set Data Type in SMT

Set data type implementation:

```
(define-sort Set (A) (Array A Bool))

(define-fun new () (Set Int)
  (lambda ((x Int)) false))

(define-fun add ((s (Set Int)) (x Int)) (Set Int)
  (store s x true))

(define-fun remove ((s (Set Int)) (x Int)) (Set Int)
  (store s x false))

(define-fun contains ((s (Set Int)) (x Int)) Bool
  (select s x))

(define-fun map ((f (Array Int Int)) (s (Set Int))) (Set Int)
  (lambda ((x Int))
    (exists ((y Int))
      (and
        (contains s y)
        (= (select f y) x)))))
```

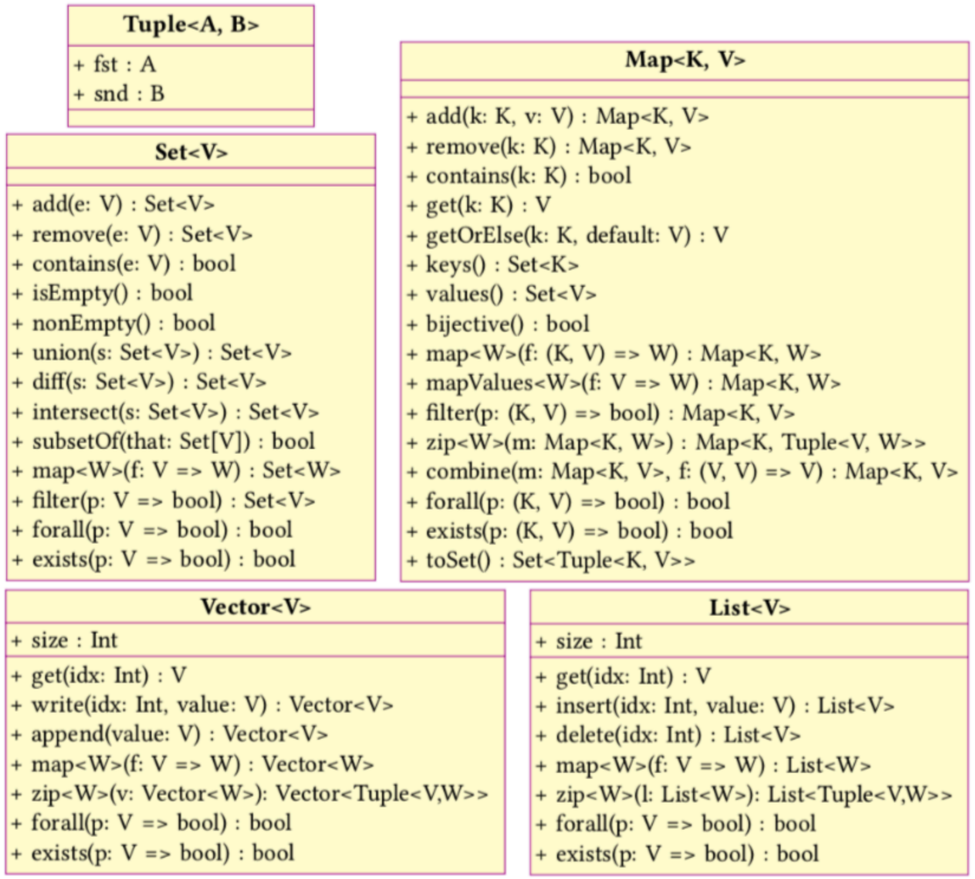
Let's verify our `map` function:

```
(declare-const s1 (Set Int)) ; some initial set
(declare-const s2 (Set Int)) ; mapped set
(declare-const s3 (Set Int)) ; verification set

; s2 = s1.map(x => x+1)
(assert
  (= s2
     (map (lambda ((x Int)) (+ x 1)) s1)))

; s3 = expected outcome
(assert
  (forall ((x Int))
    (and
      (=> (contains s1 x) (contains s3 (+ x 1)))
      (=> (not (contains s1 x)) (not (contains s3 (+ x 1)))))))

(assert (not (= s2 s3)))

(check-sat) ; outputs unsat -> no counterexamples exist
```

- SMT code is low-level —> not suited to implement complex programs
- Use Intermediate Verification Languages (IVLs) instead
  - use SMT solvers under the hood

The VeriFx Language

- High-level OOP language with extensive functional collections
  - maps, sets, vectors, etc.
- Features a novel *proof construct*
  - describes application-specific correctness properties
- No for/while/… loop constructs
  - bc problematic for automated verification
  - instead use the collections' operations

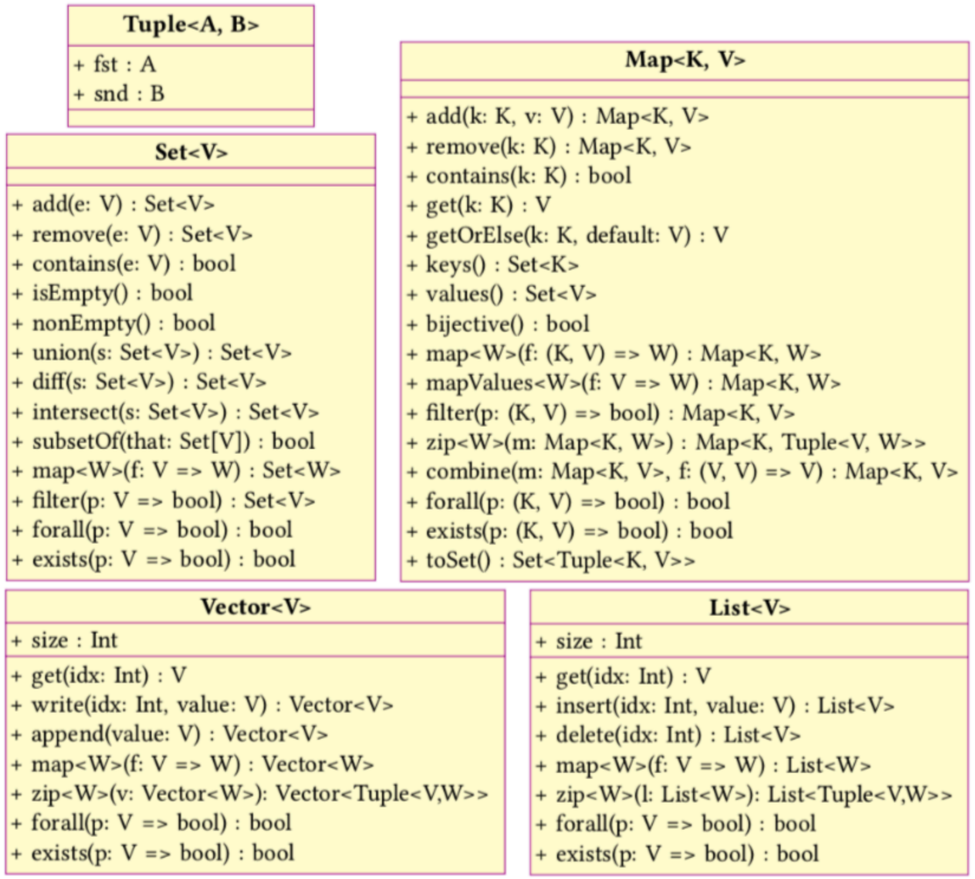


The VeriFx Language

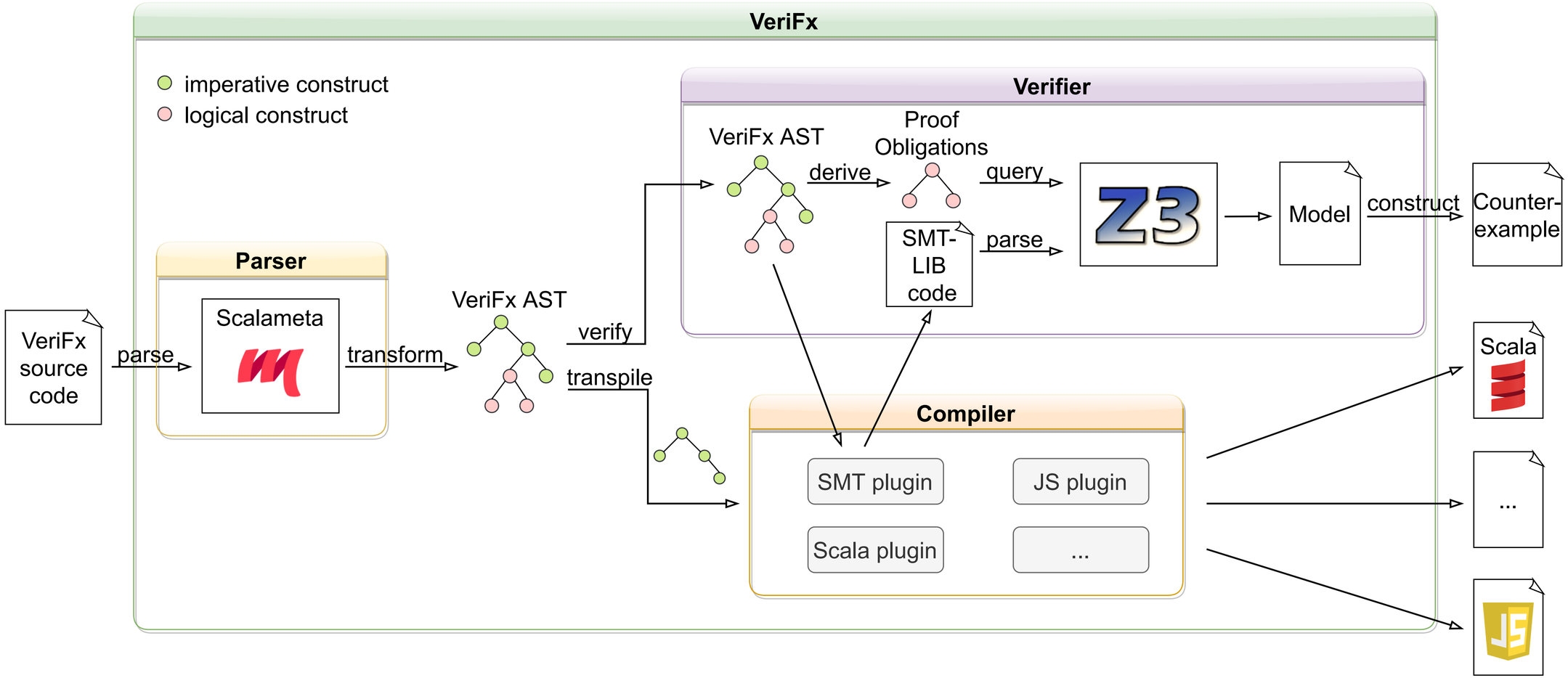
- Collections are encoded in CAL
  - CAL is decidable
  - but.. our encodings use quantifiers
    —> makes them undecidable

- Entire language is compilable to SMT
  —> Automated proof verification

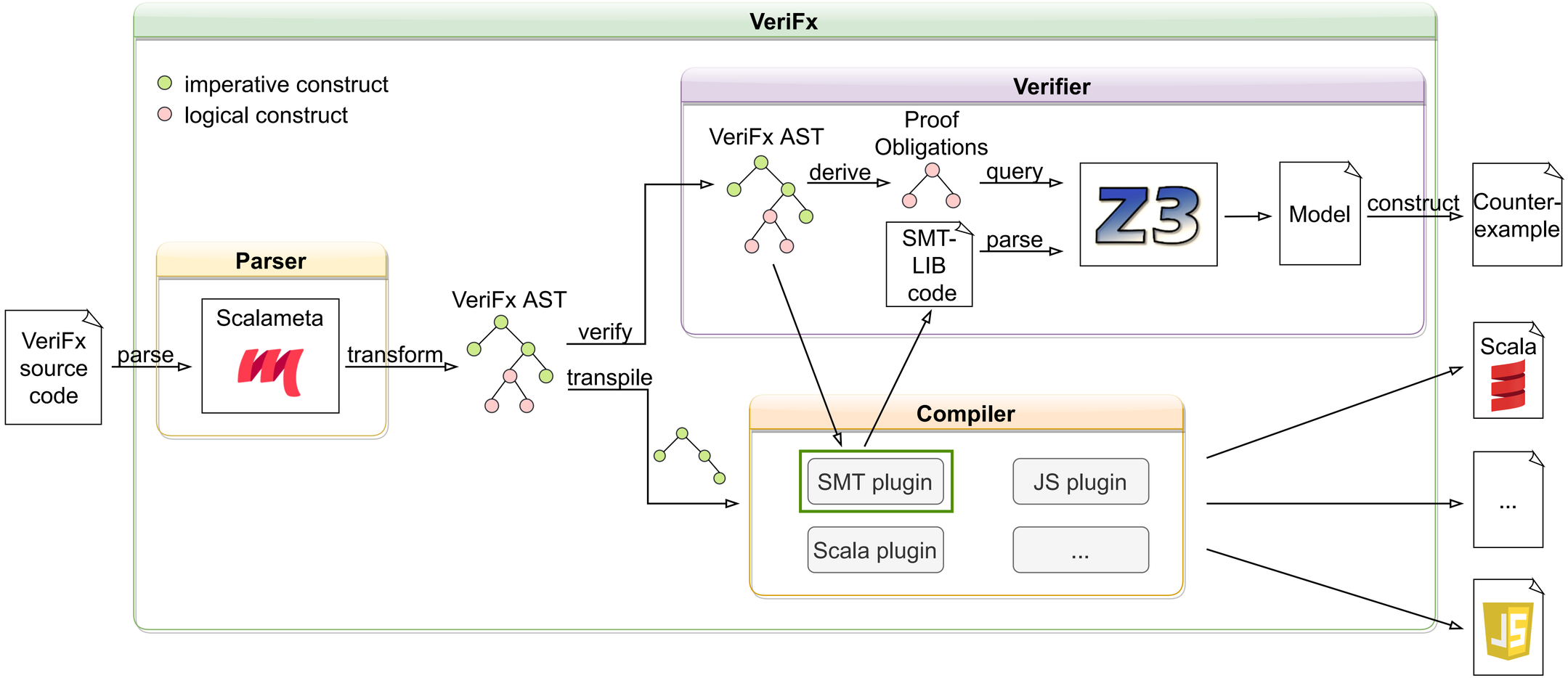
- In general, proofs are undecidable
  - but able to verify many proofs in practice



VeriFx's Architecture



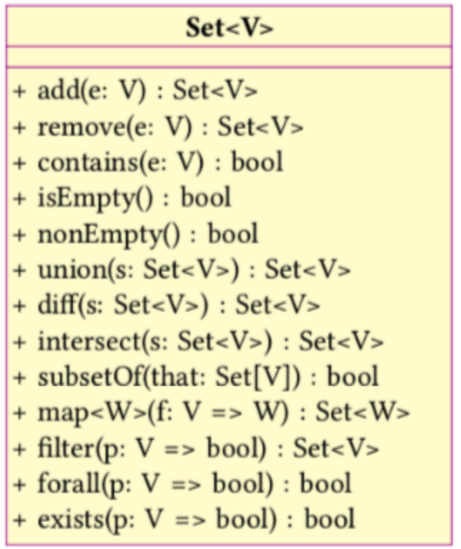
VeriFx's Architecture



VeriFx's SMT Encodings

Set data type encoding:

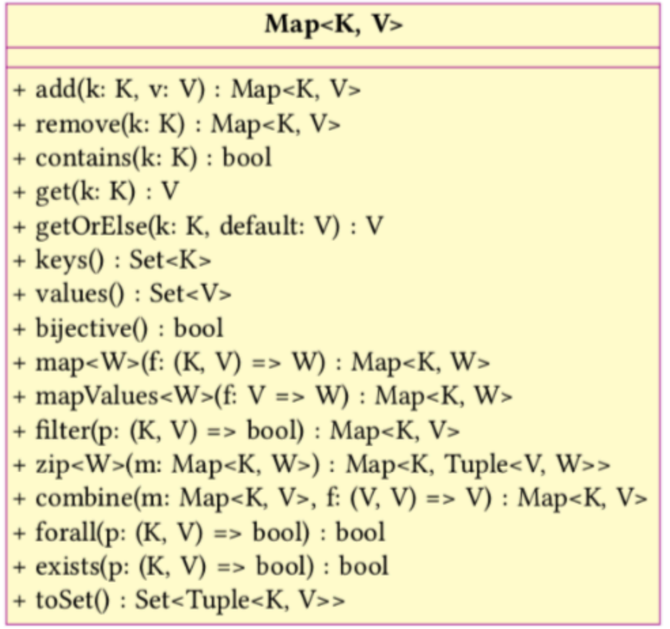
```
(define-sort Set (A) (Array A Bool))

(define-fun new () (Set Int)
  (lambda ((x Int)) false))

(define-fun add ((s (Set Int)) (x Int)) (Set Int)
  (store s x true))

(define-fun remove ((s (Set Int)) (x Int)) (Set Int)
  (store s x false))

(define-fun contains ((s (Set Int)) (x Int)) Bool
  (select s x))

(define-fun map ((f (Array Int Int)) (s (Set Int))) (Set Int)
  (lambda ((x Int))
    (exists ((y Int))
      (and
        (contains s y)
        (= (select f y) x)))))
```



VeriFx's SMT Encodings

Map data type encoding:

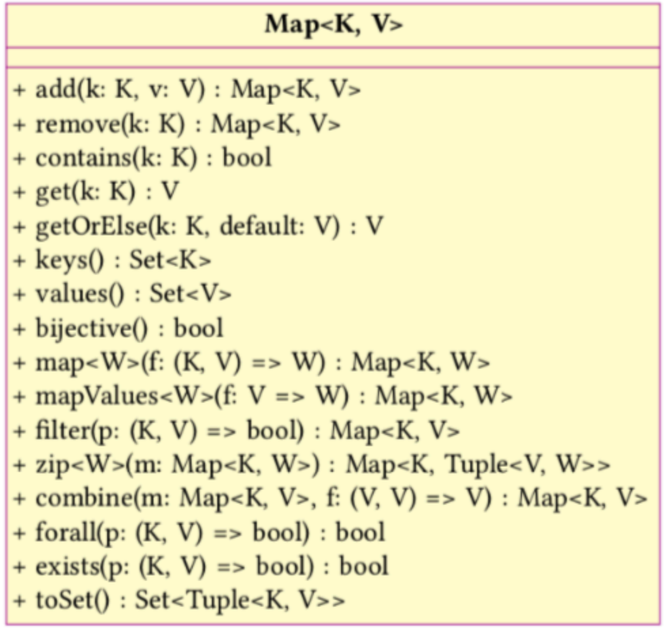
```
(define-sort Map (K V) (Array ? ?))
```



VeriFx's SMT Encodings

Map data type encoding:

```
(define-sort Map (K V) (Array K ?))
```



VeriFx's SMT Encodings

Map data type encoding:

```
(define-sort Map (K V) (Array K V))
```

Can't encode absence of a key-value pair

VeriFx's SMT Encodings

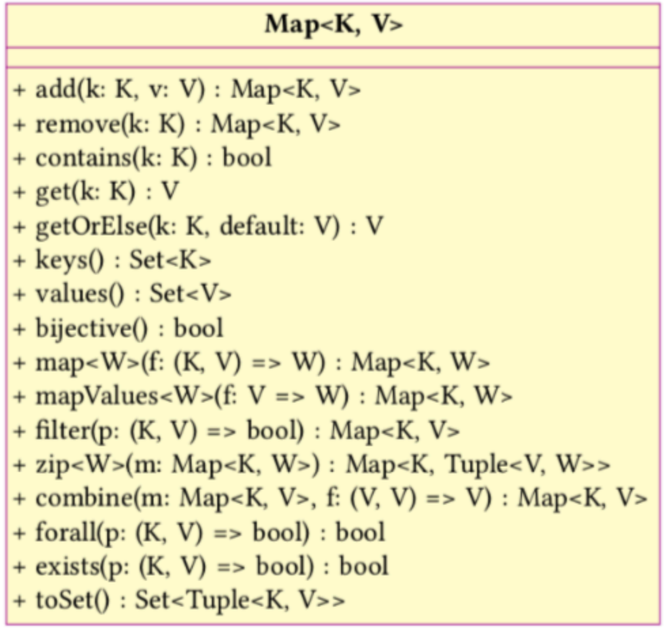
Map data type encoding:

```
(declare-datatypes (T) ((Option None (Some (get T)))))

(define-sort Map (K V) (Array K (Option V)))
```

VeriFx's SMT Encodings

Map data type encoding:

```
(declare-datatypes (T) ((Option None (Some (get T)))))

(define-sort Map (K V) (Array K (Option V)))

(define-fun new () (Map String Int)
  (lambda ((key String)) None))

(define-fun add ((m (Map String Int))(key String) (value Int)) (Map String Int)
  (store m key (Some value)))

(define-fun remove ((m (Map String Int)) (key String)) (Map String Int)
  (store m key None))

(define-fun contains ((m (Map String Int)) (key String)) Bool
  (not (= (select m key) None)))

(define-fun get ((m (Map String Int)) (key String)) Int
  (get (select m key)))
```



VeriFx's SMT Encodings

Map data type encoding:

```
(declare-datatypes (T) ((Option None (Some (get T)))))

(define-sort Map (K V) (Array K (Option V)))

(define-fun new () (Map String Int)
  (lambda ((key String)) None))

(define-fun add ((m (Map String Int))(key String) (value Int)) (Map String Int)
  (store m key (Some value)))

(define-fun remove ((m (Map String Int)) (key String)) (Map String Int)
  (store m key None))

(define-fun contains ((m (Map String Int)) (key String)) Bool
  (not (= (select m key) None)))

(define-fun get ((m (Map String Int)) (key String)) Int
  (get (select m key)))

(define-fun filter ((m (Map String Int)) (f (Array String Int Bool))) (Map String Int)
  (lambda ((key String))
    (let ((value (get m key)))
      (if (and (contains m key)
               (select f key value)) ; f(key, value) is true
          (Some value) ; then branch
          None)))) ; else branch
```

Map<K, V>
+ add(k: K, v: V) : Map<K, V> + remove(k: K) : Map<K, V> + contains(k: K) : bool + get(k: K) : V + getOrElse(k: K, default: V) : V + keys() : Set<K> + values() : Set<V> + bijective() : bool + map<W>(f: (K, V) => W) : Map<K, W> + mapValues<W>(f: V => W) : Map<K, W> + filter(p: (K, V) => bool) : Map<K, V> + zip<W>(m: Map<K, W>) : Map<K, Tuple<V, W>> + combine(m: Map<K, V>, f: (V, V) => V) : Map<K, V> + forall(p: (K, V) => bool) : bool + exists(p: (K, V) => bool) : bool + toSet() : Set<Tuple<K, V>>

VeriFx: Primitive Types and Variables

```
// Primitive types are Boolean, Int, and String
true
!false
true && false
true || false

1
1 + 2
1 - 1
2 * 3
4 / 2

"foo"

// Variables
val name: String = "Kevin" // can't be reassigned

// Types can be omitted thanks to type inference:
val age = 28
val hobby = "biking"
```

VeriFx: Collections

```
// Built-in collections are Tuple[A, B], Set[V], Map[K, V], Vector[V], LList[V]
val tuple: Tuple[String, Int] = new Tuple(name, age)
tuple.fst
tuple.snd

// All variables and data types are immutable:
val set = new Set[String]()
val set2 = set.add("foo") // creates a new set, does not mutate `set`!

val map = new Map[String, Int]()
val ages = map.add(name, age)

new Vector[String](1).write(0, "bar").write(0, "foo") // Vector("foo")
new LList[String](0).insert(0, "bar").insert(0, "foo") // LList("foo", “bar")
```

VeriFx: Functions and if tests

```
// Anonymous function
(x: Int) => x+1

// Functions are first class values
val sum: (Int, Int) => Int = (a: Int, b: Int) => a+b
val apply = (a: Int, b: Int, f: (Int, Int) => Int) => {
  f(a, b)
}
val res = apply(1, 2, sum)

// If test is an expression
// !! both branches are mandatory !!
// braces are optional if branch is a 1-liner
val minorOrMajor = if (age >= 18) "adult" else "minor"
```

VeriFx: Algebraic Data Types and Pattern Matching

```
// Algebraic data types (ADTs)
enum PrimaryColor {
  Red() | Blue() | Yellow()
}

// ADT constructors can hold data
enum Command {
  Increment(x: Int) | Decrement(x: Int) | Sum(a: Int, b: Int) | Noop()
}

// ADTs can be generic
enum SetCommand[V] {
  Add(elem: V) | Remove(elem: V)
}

// We can instantiate ADT constructors
val cmd: Command = new Increment(5)

// We can pattern match against ADTs
// Every pattern match must be exhaustive!
cmd match {
  case Increment(x) => x+1
  case Decrement(_) => 1
  case sum: Sum => sum.a + sum.b
  case _ => 0 // default case
}
```

VeriFx: Classes

```
// We can define classes with fields and methods
// within the class definition `this` refers to the current instance.
// Fields are immutable.
class Person(name: String, hobbies: Set[String]) {
  def setName(newName: String): Person = {
    new Person(newName, this.hobbies)
  }

  def addHobby(h: String) = {
    val extendedHobbies = this.hobbies.add(h)
    new Person(this.name, extendedHobbies)
  }

  def enjoys(h: String) = {
    this.hobbies.contains(h)
  }
}

// Class methods can be turned into functions by means of eta expansion
val kevin = new Person("Kevin", new Set[String]())
val addHobby: String => Person = kevin.addHobby _
val kevinWithNewHobby = addHobby("biking")
```

VeriFx: Generic Classes and Objects

```
// Classes and their methods can be generic
class WrappedSet[V](elements: Set[V]) {
  def map[W](f: V => W): WrappedSet[W] = {
    new WrappedSet(this.elements.map(f))
  }
}

// We can define singleton objects
object SumObject {
  // Objects can also contain methods.
  // Recursive methods need annotation.
  // 🐊 BEWARE: recursive methods break automated verification 🐊
  @recursive
  def sum(i: Int, res: Int): Int = {
    if (i <= 0)
      res
    else
      this.sum(i - 1, res + i)
  }
}
```

VeriFx: Proofs

```
class Person(name: String, hobbies: Set[String]) {
  def setName(newName: String): Person = {
    new Person(newName, this.hobbies)
  }

  def addHobby(h: String) = {
    val extendedHobbies = this.hobbies.add(h)
    new Person(this.name, extendedHobbies)
  }

  def enjoys(h: String) = {
    this.hobbies.contains(h)
  }
}
```

```
object Person {
  // Objects can contain proofs
  proof addHobbyIsCorrect {
    // proofs can use universal and existential quantifiers
    // by using `forall` and `exists` respectively
    forall (p: Person, h: String) {
      p.addHobby(h).enjoys(h)
    }
  }

  proof enjoysIsCorrect {
    forall (p: Person, h: String) {
      // logical implication using `=>:`
      p.hobbies.contains(h) =>: p.enjoys(h)
    }
  }
}
```

VeriFx: Generic Proofs and Counterexamples

```
object SetProofs {
  // proofs can also be generic
  proof setAddCommutes[V] {
    forall (a: V, b: V, set: Set[V]) {
      set.add(a).add(b) == set.add(b).add(a)
    }
  }

  proof setAddRemoveCommute[V] {
    forall (a: V, b: V, set: Set[V]) {
      set.add(a).remove(b) == set.remove(b).add(a)
    }
  }
}
```

VeriFx: Generic Proofs and Counterexamples

```
object SetProofs {
  // proofs can also be generic
  proof setAddCommutes[V] {
    forall (a: V, b: V, set: Set[V]) {
      set.add(a).add(b) == set.add(b).add(a)
    }
  }

  proof setAddRemoveCommute[V] {
    forall (a: V, b: V, set: Set[V]) {
      set.add(a).remove(b) == set.remove(b).add(a)
    }
  }
}
```

Counterexample:

```
Type definitions:

enum V {
  V_val_0
}

Variable assignments:

val a: V = V_val_0
val b: V = V_val_0
val set: Set[V] = Set())
```

VeriFx: Modules

src/main/verifx/be/kdeporre/app/Person.vfx

```
class Person(name: String, hobbies: Set[String]) {
  def setName(newName: String): Person = {
    new Person(newName, this.hobbies)
  }

  def addHobby(h: String) = {
    val extendedHobbies = this.hobbies.add(h)
    new Person(this.name, extendedHobbies)
  }

  def enjoys(h: String) = {
    this.hobbies.contains(h)
  }
}
```

src/main/verifx/be/kdeporre/app/Proofs.vfx

```
import be.kdeporre.app.Person

object Proofs {
  // Objects can contain proofs
  proof addHobbyIsCorrect {
    // proofs can use universal and existential quantifiers
    // by using `forall` and `exists` respectively
    forall (p: Person, h: String) {
      p.addHobby(h).enjoys(h)
    }
  }

  proof enjoysIsCorrect {
    forall (p: Person, h: String) {
      // logical implication using `=>:`
      p.hobbies.contains(h) =>: p.enjoys(h)
    }
  }
}
```

CRDT Verification

- State-based CRDTs:
  - replicas periodically send their full state to everyone
  - replicas merge incoming states
  - *merge function must be idempotent, commutative, associative*

CRDT Verification

- State-based CRDTs:
  - replicas periodically send their full state to everyone
  - replicas merge incoming states
  - *merge function must be idempotent, commutative, associative*

- Operation-based CRDTs:
  - replica applies operation locally
  - then propagates to everyone
  - *concurrent operations must commute*

Implementing State-based CRDTs in VeriFx

CvRDT framework:

```
// T is the type of the concrete implementation, e.g. `GCounter`
trait CvRDT[T <: CvRDT[T]] {
  // Excludes states that are not reachable in practice.
  // Returns true if the state is reachable, false otherwise.
  def reachable(): Boolean = true

  // Excludes replicas that are not compatible
  // e.g. OptORSet keeps a summary vector
  //      every replica must have a summary vector of the same size
  //      because it assumes a fixed number of replicas!
  def compatible(that: T): Boolean = true

  // Returns the LUB of `this` and `that`
  def merge(that: T): T

  // True if `this <= that`
  def compare(that: T): Boolean

  // Default implementation of `equals`.
  // `a equals b` iff `a <= b && b <= a` according to `compare`.
  def equals(that: T): Boolean =
    this.asInstanceOf[T].compare(that) && that.compare(this.asInstanceOf[T])
}
```

Verifying State-based CRDTs in VeriFx

CvRDT proof trait:

```
trait CvRDTProof[T <: CvRDT[T]] {
  proof is_a_CvRDT {
    forall (x: T, y: T, z: T) {
      ( x.reachable() && y.reachable() && z.reachable() &&
          x.compatible(y) && x.compatible(z) && y.compatible(z) ) =>: {
        x.merge(x).equals(x) && // idempotent
        x.merge(y).equals(y.merge(x)) && // commutative
        x.merge(y).merge(z).equals(x.merge(y.merge(z))) && // associative
        x.merge(y).reachable() && // merged state is reachable
        x.merge(y).merge(z).reachable() &&
        x.compatible(y) == y.compatible(x) // compatible commutes
      }
    }
  }

  proof compareCorrect {
    forall (x: T, y: T) {
      x.equals(y) == (x == y)
    }
  }
}
```

Verifying State-based CRDTs in VeriFx

CvRDT proof trait:

```
trait CvRDTProof[T <: CvRDT[T]] {
  proof mergeCommutative {
    forall (x: T, y: T) {
      (x.reachable() && y.reachable() && x.compatible(y)) =>: {
        x.merge(y).equals(y.merge(x)) &&
        x.merge(y).reachable()
      }
    }
  }

  proof mergeIdempotent {
    forall (x: T) {
      x.reachable() =>: x.merge(x).equals(x)
    }
  }

  proof mergeAssociative {
    forall (x: T, y: T, z: T) {
      ( x.reachable() && y.reachable() && z.reachable() &&
          x.compatible(y) && x.compatible(z) && y.compatible(z) ) =>: {
        x.merge(y).merge(z).equals(x.merge(y.merge(z))) &&
        x.merge(y).merge(z).reachable()
      }
    }
  }
}
```

Verifying State-based CRDTs in VeriFx

CvRDT proof trait:

```
trait CvRDTProof[T <: CvRDT[T]] {
  proof mergeCommutative {
    forall (x: T, y: T) {
      (x.reachable() && y.reachable() && x.compatible(y)) =>: {
        x.merge(y).equals(y.merge(x)) &&
        x.merge(y).reachable()
      }
    }
  }

  proof mergeIdempotent {
    forall (x: T) {
      x.reachable() =>: x.merge(x).equals(x)
    }
  }

  proof mergeAssociative {
    forall (x: T, y: T, z: T) {
      ( x.reachable() && y.reachable() && z.reachable() &&
          x.compatible(y) && x.compatible(z) && y.compatible(z) ) =>: {
        x.merge(y).merge(z).equals(x.merge(y.merge(z))) &&
        x.merge(y).merge(z).reachable()
      }
    }
  }
}
```

Verifying Generic State-based CRDTs in VeriFx

CvRDT proof trait:

```
trait CvRDTProof1[ T[A] <: CvRDT[T[A]] ] {

  proof is_a_CvRDT[S] {
    forall (x: T[S], y: T[S], z: T[S]) {
      ( x.reachable() && y.reachable() && z.reachable() &&
        x.compatible(y) && x.compatible(z) && y.compatible(z) ) =>: {
        x.merge(x).equals(x) && // idempotent
        x.merge(y).equals(y.merge(x)) && // commutative
        x.merge(y).merge(z).equals(x.merge(y.merge(z))) && // associative
        x.merge(y).reachable() && // merged state is reachable
        x.merge(y).merge(z).reachable() &&
        x.compatible(y) == y.compatible(x) // compatible commutes
      }
    }
  }

  // ...
}
```

Verifying Generic State-based CRDTs in VeriFx

CvRDT proof trait:

```
trait CvRDTProof1[ T[A] <: CvRDT[T[A]] ] {

  proof is_a_CvRDT[S] {
    forall (x: T[S], y: T[S], z: T[S]) {
      ( x.reachable() && y.reachable() && z.reachable() &&
        x.compatible(y) && x.compatible(z) && y.compatible(z) ) =>: {
        x.merge(x).equals(x) && // idempotent
        x.merge(y).equals(y.merge(x)) && // commutative
        x.merge(y).merge(z).equals(x.merge(y.merge(z))) && // associative
        x.merge(y).reachable() && // merged state is reachable
        x.merge(y).merge(z).reachable() &&
        x.compatible(y) == y.compatible(x) // compatible commutes
      }
    }
  }

  // ...
}
```

Grow-only Counter CRDT in VeriFx

Specification 6 State-based increment-only counter (vector version)

```
1:  payload integer[n] P                                  ▷ One entry per replica
2:      initial [0, 0, ..., 0]
3:  update increment ()
4:      let g = myID()                                    ▷ g: source replica
5:      P[g] := P[g] + 1
6:  query value () : integer v
7:      let v = Σ_i P[i]
8:  compare (X, Y) : boolean b
9:      let b = (∀i ∈ [0, n − 1] : X.P[i] ≤ Y.P[i])
10: merge (X, Y) : payload Z
11:     let ∀i ∈ [0, n − 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. [Research Report] RR-7506, Inria – Centre Paris-Rocquencourt; INRIA. 2011, pp.50. inria-00555588

Grow-only Counter CRDT in VeriFx

Specification 6 State-based increment-only counter (vector version)

payload integer$[n]$ P ▷ One entry per replica
    initial $[0, 0, \ldots, 0]$
update *increment* ()
    let $g = myID()$ ▷ g: source replica
    $P[g] := P[g] + 1$
query *value* () : integer v
    let $v = \sum_i P[i]$
compare (X, Y) : boolean b
    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
merge (X, Y) : payload Z
    let $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$

```
class GCounter(entries: Vector[Int]) extends CvRDT[GCounter] {
  def increment(replica: Int) = {
    val count = this.entries.get(replica)
    new GCounter(this.entries.write(replica, count + 1))
  }

  def value() = this.computeValue()

  @recursive
  private def computeValue(sum: Int = 0, replica: Int = 0): Int = {
    if (replica >= 0 && replica < this.entries.size) {
      val count = this.entries.get(replica)
      this.computeValue(sum + count, replica + 1)
    }
    else sum
  }

  def merge(that: GCounter): GCounter = {
    val max = (t: Tuple[Int, Int]) => if (t.fst >= t.snd) t.fst else t.snd
    val mergedEntries = this.entries.zip(that.entries).map(max)
    new GCounter(mergedEntries)
  }

  def compare(that: GCounter): Boolean = {
    this.entries.zip(that.entries)
      .forall((tup: Tuple[Int, Int]) => tup.fst <= tup.snd)
  }
}
```

Grow-only Counter CRDT in VeriFx

Specification 6 State-based increment-only counter (vector version)

1: payload integer$[n]$ P ▷ One entry per replica
2: initial $[0, 0, \ldots, 0]$
3: update *increment* ()
4: let $g = myID()$ ▷ g: source replica
5: $P[g] := P[g] + 1$
6: query *value* () : integer v
7: let $v = \sum_i P[i]$
8: compare (X, Y) : boolean b
9: let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
10: merge (X, Y) : payload Z
11: let $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$

```
class GCounter(entries: Vector[Int]) extends CvRDT[GCounter] {
  def increment(replica: Int) = {
    val count = this.entries.get(replica)
    new GCounter(this.entries.write(replica, count + 1))
  }

  def value() = this.computeValue()

  @recursive
  private def computeValue(sum: Int = 0, replica: Int = 0): Int = {
    if (replica >= 0 && replica < this.entries.size) {
      val count = this.entries.get(replica)
      this.computeValue(sum + count, replica + 1)
    }
    else sum
  }

  def merge(that: GCounter): GCounter = {
    val max = (t: Tuple[Int, Int]) => if (t.fst >= t.snd) t.fst else t.snd
    val mergedEntries = this.entries.zip(that.entries).map(max)
    new GCounter(mergedEntries)
  }

  def compare(that: GCounter): Boolean = {
    this.entries.zip(that.entries)
      .forall((tup: Tuple[Int, Int]) => tup.fst <= tup.snd)
  }
}
```

```
object GCounter extends CvRDTProof[GCounter]
```

Grow-only Counter CRDT in VeriFx

Specification 6 State-based increment-only counter (vector version)

payload integer$[n]$ P ▷ One entry per replica
    initial $[0, 0, \ldots, 0]$
update *increment* ()
    let $g = myID()$ ▷ g: source replica
    $P[g] := P[g] + 1$
query *value* () : integer v
    let $v = \sum_i P[i]$
compare (X, Y) : boolean b
    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
merge (X, Y) : payload Z
    let $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$

```
class GCounter(entries: Vector[Int]) extends CvRDT[GCounter] {
  def increment(replica: Int) = {
    val count = this.entries.get(replica)
    new GCounter(this.entries.write(replica, count + 1))
  }

  def value() = this.computeValue()

  @recursive
  private def computeValue(sum: Int = 0, replica: Int = 0): Int = {
    if (replica >= 0 && replica < this.entries.size) {
      val count = this.entries.get(replica)
      this.computeValue(sum + count, replica + 1)
    }
    else sum
  }

  def merge(that: GCounter): GCounter = {
    val max = (t: Tuple[Int, Int]) => if (t.fst >= t.snd) t.fst else t.snd
    val mergedEntries = this.entries.zip(that.entries).map(max)
    new GCounter(mergedEntries)
  }

  def compare(that: GCounter): Boolean = {
    this.entries.zip(that.entries)
      .forall((tup: Tuple[Int, Int]) => tup.fst <= tup.snd)
  }
}
```

```
object GCounter extends CvRDTProof[GCounter]

 - GCounter.is_a_CvRDT [proved]
 - GCounter.mergeIdempotent [proved]
 - GCounter.mergeCommutative [proved]
 - GCounter.mergeAssociative [proved]
 - GCounter.compareCorrect [proved]
```

Implementing Operation-based CRDTs in VeriFx

CmRDT framework:

```
trait CmRDT[Op, Msg, T <: CmRDT[Op, Msg, T]] {
  def reachable(): Boolean = true
  def compatible(x: Msg, y: Msg): Boolean = true
  def compatibleS(that: T): Boolean = true

  def enabledSrc(op: Op): Boolean = true
  def enabledDown(msg: Msg): Boolean = true

  def prepare(op: Op): Msg
  def effect(msg: Msg): T

  // Applies an incoming message (downstream) but only if its downstream precondition holds
  def tryEffect(msg: Msg): T = {
    if (this.enabledDown(msg))
      this.effect(msg)
    else
      this.asInstanceOf[T]
  }

  def equals(that: T): Boolean = {
    this == that
  }
}
```

Verifying Operation-based CRDTs in VeriFx

CmRDT proof trait:

```
trait CmRDTProof[Op, Msg, T <: CmRDT[Op, Msg, T]] {
  proof is_a_CmRDT {
    forall (s1: T, s2: T, s3: T, x: Op, y: Op) {
      // Invoke 2 operations concurrently
      val msg1 = s1.prepare(x) // replica s1 locally invokes operation `x` which prepares a message
      val msg2 = s2.prepare(y) // replica s2 locally invokes operation `y` which prepares a message

      ( s1.reachable() && s2.reachable() && s3.reachable() &&
        s1.enabledSrc(x) && s2.enabledSrc(y) &&
        s1.compatible(msg1, msg2) && s1.compatibleS(s2) && s1.compatibleS(s3) && s2.compatibleS(s3) ) =>: {

        // The effectors must commute
        s3.tryEffect(msg1).tryEffect(msg2).equals(s3.tryEffect(msg2).tryEffect(msg1)) &&
        // The intermediate and resulting states must be reachable,
        // if not there is an error in the definition of `reachable`
        s3.tryEffect(msg1).reachable() &&
        s3.tryEffect(msg2).reachable() &&
        s3.tryEffect(msg1).tryEffect(msg2).reachable()
      }
    }
  }
}
```

Operation-based Counter CRDT in VeriFx

Specification 5 op-based Counter

```
1:  payload integer i
2:      initial 0
3:  query value () : integer j
4:      let j = i
5:  update increment ()
6:      downstream ()                          ▷ No precond: delivery order is empty
7:          i := i + 1
8:  update decrement ()
9:      downstream ()                          ▷ No precond: delivery order is empty
10:         i := i − 1
```

Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. [Research Report] RR-7506, Inria – Centre Paris-Rocquencourt; INRIA. 2011, pp.50. inria-00555588

Operation-based Counter CRDT in VeriFx

```
object CounterOps {
  enum Op {
    Inc() | Dec()
  }
}

class Counter(v: Int = 0) extends CmRDT[Op, Op, Counter] {
  def increment() = new Counter(this.v + 1)
  def decrement() = new Counter(this.v - 1)

  def prepare(op: Op) = op // prepare phase does not add extra information

  def effect(op: Op) = op match {
    case Inc() => this.increment()
    case Dec() => this.decrement()
  }
}
```

Specification 5 op-based Counter

```
payload integer i
    initial 0
query value () : integer j
    let j = i
update increment ()
    downstream ()
        i := i + 1
update decrement ()
    downstream ()
        i := i − 1
```

Operation-based Counter CRDT in VeriFx

```
object CounterOps {
  enum Op {
    Inc() | Dec()
  }
}

class Counter(v: Int = 0) extends CmRDT[Op, Op, Counter] {
  def increment() = new Counter(this.v + 1)
  def decrement() = new Counter(this.v - 1)

  def prepare(op: Op) = op // prepare phase does not add extra information

  def effect(op: Op) = op match {
    case Inc() => this.increment()
    case Dec() => this.decrement()
  }
}

object Counter extends CmRDTProof[Op, Op, Counter]
```

Specification 5 op-based Counter

```
payload integer i
    initial 0
query value () : integer j
    let j = i
update increment ()
    downstream ()
        i := i + 1
update decrement ()
    downstream ()
        i := i − 1
```

Operation-based Counter CRDT in VeriFx

```
object CounterOps {
  enum Op {
    Inc() | Dec()
  }
}

class Counter(v: Int = 0) extends CmRDT[Op, Op, Counter] {
  def increment() = new Counter(this.v + 1)
  def decrement() = new Counter(this.v - 1)

  def prepare(op: Op) = op // prepare phase does not add extra information

  def effect(op: Op) = op match {
    case Inc() => this.increment()
    case Dec() => this.decrement()
  }
}

object Counter extends CmRDTProof[Op, Op, Counter]
```

```
- Counter.is_a_CmRDT [proved]
```

Specification 5 op-based Counter

```
1:  payload integer i
2:      initial 0
3:  query value () : integer j
4:      let j = i
5:  update increment ()
6:      downstream ()
7:          i := i + 1
8:  update decrement ()
9:      downstream ()
10:         i := i − 1
```

Project Suggestions

- Show equivalence between well-known state- and op-based CRDTs. You can start from CRDTs we already implemented but you will need to verify equivalence
- Implement and verify delta CRDTs seen in Baquero's lecture and from his original paper
- Implement and verify tombstone-based Operational Transformation functions
- Apply VeriFx to a domain of your choice (e.g. blockchain) or that may be relevant to your master/phd research