Teaching with Logika

Conceiving and Constructing Correct Software

Stefan Hallerstede[1] (sha@ece.au.dk)
John Hatcliff[2] (hatcliff@ksu.edu)
Robby[2] (robby@ksu.edu)

[1]Aarhus University
[2]Kansas State University

FMTea, Milan, Italy, 10 September 2024

Context, Approach and Evolution

Use and Significance of Slang and Logika

Feedback

Discussion

Next Steps

Software Engineering Curriculum

- Development of *new BSc/MSc curriculum* at Aarhus University (Engineering, Fall 2018)
- *Problem solving*, *modelling*, *reasoning*, and *verification* are woven into "common" courses

Software Engineering Curriculum

- Development of *new BSc/MSc curriculum* at Aarhus University (Engineering, Fall 2018)
- *Problem solving*, *modelling*, *reasoning*, and *verification* are woven into "common" courses
  - Introduction to Programming (BSc 10 ECTS – informal)
  - Software Architecture (BSc 5 ECTS – informal)
  - Discrete Mathematics (BSc 5 ECTS – informal)
  - *Programming and Modelling* (BSc 10 ECTS – formal)
  - Declarative Programming (BSc&MSc 10 ECTS – informal/formal)
  - **Software Correctness** (MSc 5 ECTS – formal)

Software Engineering Curriculum

- Development of *new BSc/MSc curriculum* at Aarhus University (Engineering, Fall 2018)
- *Problem solving*, *modelling*, *reasoning*, and *verification* are woven into "common" courses
  - Introduction to Programming (BSc 10 ECTS – informal)
  - Software Architecture (BSc 5 ECTS – informal)
  - Discrete Mathematics (BSc 5 ECTS – informal)
  - *Programming and Modelling* (BSc 10 ECTS – formal)
  - Declarative Programming (BSc&MSc 10 ECTS – informal/formal)
  - **Software Correctness** (MSc 5 ECTS – formal)
- Local students are prepared for formal methods thinking
- They see **Slang** and **Logika** in *Programming and Modelling* and **Software Correctness**
  - **Slang**: Scala dialect with verification support
  - **Logika**: Interactive support for programming and verifying with Slang

Software Engineering Curriculum

- Development of *new BSc/MSc curriculum* at Aarhus University (Engineering, Fall 2018)
- *Problem solving*, *modelling*, *reasoning*, and *verification* are woven into "common" courses
  - Introduction to Programming (BSc 10 ECTS – informal)
  - Software Architecture (BSc 5 ECTS – informal)
  - Discrete Mathematics (BSc 5 ECTS – informal)
  - *Programming and Modelling* (BSc 10 ECTS – formal)
  - Declarative Programming (BSc&MSc 10 ECTS – informal/formal)
  - **Software Correctness** (MSc 5 ECTS – formal)
- Local students are prepared for formal methods thinking
- They see **Slang** and **Logika** in *Programming and Modelling* and **Software Correctness**
  - **Slang**: Scala dialect with verification support
  - **Logika**: Interactive support for programming and verifying with Slang
- Cohort on MSc level is mixed – **background at MSc level varies**
  - Slang and Logika are well-suited for this situation

Evolution of the Course

- In 2012 started predecessor course (as **programming course**)
    - Using Java, Scheme and Prolog
    - **Some reasoning** about functional and logical programs
    - Used **informal inductive proofs**

Evolution of the Course

- In 2012 started predecessor course (as **programming course**)
  - Using Java, Scheme and Prolog
  - **Some reasoning** about functional and logical programs
  - Used **informal inductive proofs**
- In 2016 tried to include some **Coq**
- In 2017 tried to include some **Isabelle**
  - Both of these were **difficult to get across to the students**

Evolution of the Course

- In 2012 started predecessor course (as **programming course**)
  - Using Java, Scheme and Prolog
  - **Some reasoning** about functional and logical programs
  - Used **informal inductive proofs**
- In 2016 tried to include some **Coq**
- In 2017 tried to include some **Isabelle**
  - Both of these were **difficult to get across to the students**
- In 2018 introduced **Slang**/**Logika**
  - This worked well, staying focused on **programming** methodology
  - Started design of new course based on Slang/Logika (*dropping Prolog*)

Evolution of the Course

- In 2012 started predecessor course (as **programming course**)
  - Using Java, Scheme and Prolog
  - **Some reasoning** about functional and logical programs
  - Used **informal inductive proofs**
- In 2016 tried to include some **Coq**
- In 2017 tried to include some **Isabelle**
  - Both of these were **difficult to get across to the students**
- In 2018 introduced **Slang**/**Logika**
  - This worked well, staying focused on **programming** methodology
  - Started design of new course based on Slang/Logika (*dropping Prolog*)
- In 2023 the **new course** Software Correctness was established

Content and Objectives

- Schedule:
  - **Week 1: Introduction** – Reasoning about software (and **tool** installation)
  - **Week 2: Tracing Facts** – Pick up the students **reasoning in familiar ways**
  - **Week 3: Conditionals** – **Progress slowly** discussing different approaches
  - **Week 4: Contracts (Test)** – Ensure **students see benefit** for their programming skills
  - **Week 5: Contracts (Proof)** – Based on preceding week but using compositional **proof**
  - **Week 6: Loops and Recursion** – Some theory: programs are just another kind of formula
  - **Week 7: Unfolding and Fixpoints** – More theory with large and complex formulas
  - **Week 8: Loops and Recursion Testing** – Ensure **students see benefit**
  - **Week 9: Sequences and Arrays** – Increase complexity of programs
  - **Finally: Verification Examples and Practice** – Provide **methodology** backed by examples

Content and Objectives

- Schedule:
  - **Week 1: Introduction** – Reasoning about software (and **tool** installation)
  - **Week 2: Tracing Facts** – Pick up the students **reasoning in familiar ways**
  - **Week 3: Conditionals** – **Progress slowly** discussing different approaches
  - **Week 4: Contracts (Test)** – Ensure **students see benefit** for their programming skills
  - **Week 5: Contracts (Proof)** – Based on preceding week but using compositional **proof**
  - **Week 6: Loops and Recursion** – Some theory: programs are just another kind of formula
  - **Week 7: Unfolding and Fixpoints** – More theory with large and complex formulas
  - **Week 8: Loops and Recursion Testing** – Ensure **students see benefit**
  - **Week 9: Sequences and Arrays** – Increase complexity of programs
  - **Finally: Verification Examples and Practice** – Provide **methodology** backed by examples
  - Accompanied by a **programming project** where some test and proof is applied (mostly at home)
  - Exercises **only in class** (teacher helps)

Content and Objectives

- Schedule:
  - **Week 1: Introduction** – Reasoning about software (and **tool** installation)
  - **Week 2: Tracing Facts** – Pick up the students **reasoning in familiar ways**
  - **Week 3: Conditionals** – **Progress slowly** discussing different approaches
  - **Week 4: Contracts (Test)** – Ensure **students see benefit** for their programming skills
  - **Week 5: Contracts (Proof)** – Based on preceding week but using compositional **proof**
  - **Week 6: Loops and Recursion** – Some theory: programs are just another kind of formula
  - **Week 7: Unfolding and Fixpoints** – More theory with large and complex formulas
  - **Week 8: Loops and Recursion Testing** – Ensure **students see benefit**
  - **Week 9: Sequences and Arrays** – Increase complexity of programs
  - **Finally: Verification Examples and Practice** – Provide **methodology** backed by examples
  - Accompanied by a **programming project** where some test and proof is applied (mostly at home)
  - Exercises **only in class** (teacher helps)
- Objectives:
  - Improve **programming** skills, **testing** skills, **documentation** skills, **reasoning** skills
  - Do not limit students' vision to Slang,
    so the material becomes relevant **beyond the course**

Context, Approach and Evolution

Use and Significance of Slang and Logika

Feedback

Discussion

Next Steps

A Quick Tour of Slang and Logika

The User Interface

- Week3
  - absvalue_nn.sc
  - boolor.sc
  - maximum_diff.sc
  - maximum_diff_ded.sc
  - maximum_simple.sc
  - maximum_simple_ded.sc
  - nested_max.sc
  - swap_block.sc
  - swap_choice.sc
- Week4
  - Linear_Combination_Contract_Test_Fault.sc
  - Max_Function_Frame.sc
  - Max_Function_Frame_Contract.sc
  - Max_Function_Frame_Contract_Test.sc
  - Max_Function_Pure.sc
  - Max_Function_Pure_Contract.sc
  - Max_Function_Pure_Contract_Test.sc
  - Max_Function_Pure_Deduce.sc
  - Two_Square_Contract_Test_Fault.sc
- Week5
  - Linear_Function_Impure.sc
  - Linear_Function_Impure_Spec.sc
  - Linear_Function_Pure.sc
  - Swap_Function_Contract.sc
  - Swap_Function_Fun_Contracts.sc
  - Swap_Mutable_Assignment_Contracts.sc
  - Swap_Mutable_Assignment_Exercise.sc
  - Swap_Mutable_Assignment_Frames.sc
  - Swap_Mutable_Deduce.sc
- Week7
  - Count_Int_Loop_Rec.sc
  - Count_Int_Loop_Rec_Term.sc
  - Fac_Function_Loop_Rec.sc
  - Fac_Function_Loop_Rec_Term.sc

```
// #Sireum #Logika
import org.sireum._
val x: Z = randomInt()
val y: Z = randomInt()

var z: Z = 0
if (x < y) {


  z = y


} else {


  z = x


}
assert(z ≡ x ∨ z ≡ y)
assert(y ≤ z ∧ x ≤ z)
```

A Quick Tour of Slang and Logika

The User Interface

- Week3
  - absvalue_nn.sc
  - boolor.sc
  - maximum_diff.sc
  - maximum_diff_ded.sc
  - maximum_simple.sc
  - maximum_simple_ded.sc
  - nested_max.sc
  - swap_block.sc
  - swap_choice.sc
- Week4
  - Linear_Combination_Contract_Test_Fault.sc
  - Max_Function_Frame.sc
  - Max_Function_Frame_Contract.sc
  - Max_Function_Frame_Contract_Test.sc
  - Max_Function_Pure.sc
  - Max_Function_Pure_Contract.sc
  - Max_Function_Pure_Contract_Test.sc
  - Max_Function_Pure_Deduce.sc
  - Two_Square_Contract_Test_Fault.sc
- Week5
  - Linear_Function_Impure.sc
  - Linear_Function_Impure_Spec.sc
  - Linear_Function_Pure.sc
  - Swap_Function_Contract.sc
  - Swap_Function_Fun_Contracts.sc
  - Swap_Mutable_Assignment_Contracts.sc
  - Swap_Mutable_Assignment_Exercise.sc
  - Swap_Mutable_Assignment_Frames.sc
  - Swap_Mutable_Deduce.sc
- Week7
  - Count_Int_Loop_Rec.sc
  - Count_Int_Loop_Rec_Term.sc
  - Fac_Function_Loop_Rec.sc
  - Fac_Function_Loop_Rec_Term.sc

```
// #Sireum #Logika
import org.sireum._
val x: Z = randomInt()
val y: Z = randomInt()

var z: Z = 0
if (x < y) {


  z = y


} else {


  z = x


}
assert(z ≡ x ∨ z ≡ y)
assert(y ≤ z ∧ x ≤ z)
```

Slang is a dialect of the **Scala programming language**

A Quick Tour of Slang and Logika

The User Interface

- Week3
  - absvalue_nn.sc
  - boolor.sc
  - maximum_diff.sc
  - maximum_diff_ded.sc
  - maximum_simple.sc
  - maximum_simple_ded.sc
  - nested_max.sc
  - swap_block.sc
  - swap_choice.sc
- Week4
  - Linear_Combination_Contract_Test_Fault.sc
  - Max_Function_Frame.sc
  - Max_Function_Frame_Contract.sc
  - Max_Function_Frame_Contract_Test.sc
  - Max_Function_Pure.sc
  - Max_Function_Pure_Contract.sc
  - Max_Function_Pure_Contract_Test.sc
  - Max_Function_Pure_Deduce.sc
  - Two_Square_Contract_Test_Fault.sc
- Week5
  - Linear_Function_Impure.sc
  - Linear_Function_Impure_Spec.sc
  - Linear_Function_Pure.sc
  - Swap_Function_Contract.sc
  - Swap_Function_Fun_Contracts.sc
  - Swap_Mutable_Assignment_Contracts.sc
  - Swap_Mutable_Assignment_Exercise.sc
  - Swap_Mutable_Assignment_Frames.sc
  - Swap_Mutable_Deduce.sc
- Week7
  - Count_Int_Loop_Rec.sc
  - Count_Int_Loop_Rec_Term.sc
  - Fac_Function_Loop_Rec.sc
  - Fac_Function_Loop_Rec_Term.sc

```
// #Sireum #Logika
import org.sireum._
val x: Z = randomInt()
val y: Z = randomInt()

var z: Z = 0
if (x < y) {


  z = y


} else {


  z = x


}
assert(z == x ∨ z == y)
assert(y ≤ z ∧ x ≤ z)
```

Slang is a dialect of the **Scala programming language**

Functional and imperative programming

A Quick Tour of Slang and Logika

The User Interface

- Week3
  - absvalue_nn.sc
  - boolor.sc
  - maximum_diff.sc
  - maximum_diff_ded.sc
  - maximum_simple.sc
  - maximum_simple_ded.sc
  - nested_max.sc
  - swap_block.sc
  - swap_choice.sc
- Week4
  - Linear_Combination_Contract_Test_Fault.sc
  - Max_Function_Frame.sc
  - Max_Function_Frame_Contract.sc
  - Max_Function_Frame_Contract_Test.sc
  - Max_Function_Pure.sc
  - Max_Function_Pure_Contract.sc
  - Max_Function_Pure_Contract_Test.sc
  - Max_Function_Pure_Deduce.sc
  - Two_Square_Contract_Test_Fault.sc
- Week5
  - Linear_Function_Impure.sc
  - Linear_Function_Impure_Spec.sc
  - Linear_Function_Pure.sc
  - Swap_Function_Contract.sc
  - Swap_Function_Fun_Contracts.sc
  - Swap_Mutable_Assignment_Contracts.sc
  - Swap_Mutable_Assignment_Exercise.sc
  - Swap_Mutable_Assignment_Frames.sc
  - Swap_Mutable_Deduce.sc
- Week7
  - Count_Int_Loop_Rec.sc
  - Count_Int_Loop_Rec_Term.sc
  - Fac_Function_Loop_Rec.sc
  - Fac_Function_Loop_Rec_Term.sc

```
// #Sireum #Logika
import org.sireum._
val x: Z = randomInt()
val y: Z = randomInt()

var z: Z = 0
if (x < y) {


  z = y


} else {


  z = x


}
assert(z ≡ x ∨ z ≡ y)
assert(y ≤ z ∧ x ≤ z)
```

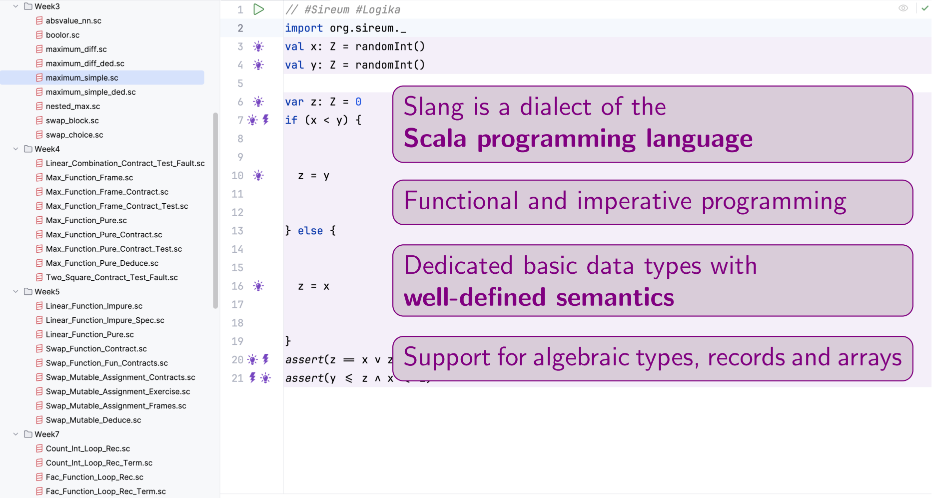
Slang is a dialect of the **Scala programming language**

Functional and imperative programming

Dedicated basic data types with **well-defined semantics**

A Quick Tour of Slang and Logika

The User Interface



Slang is a dialect of the **Scala programming language**

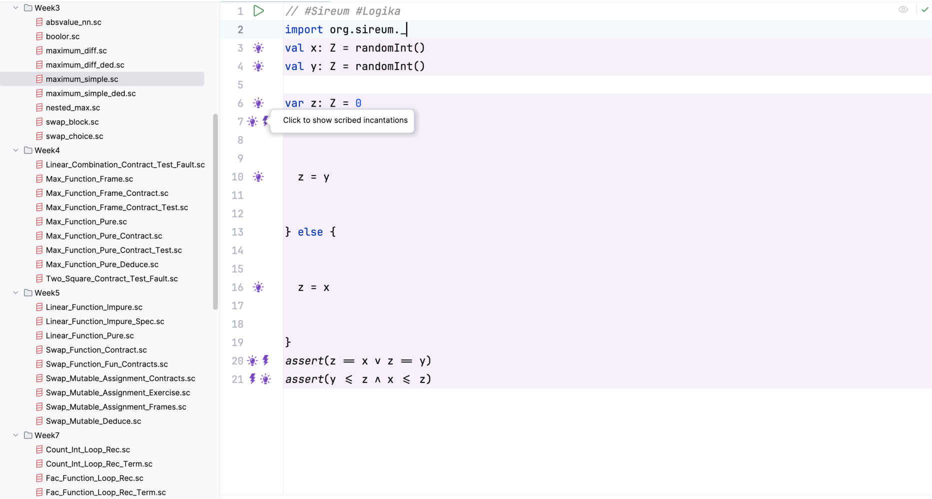
Functional and imperative programming

Dedicated basic data types with **well-defined semantics**

Support for algebraic types, records and arrays

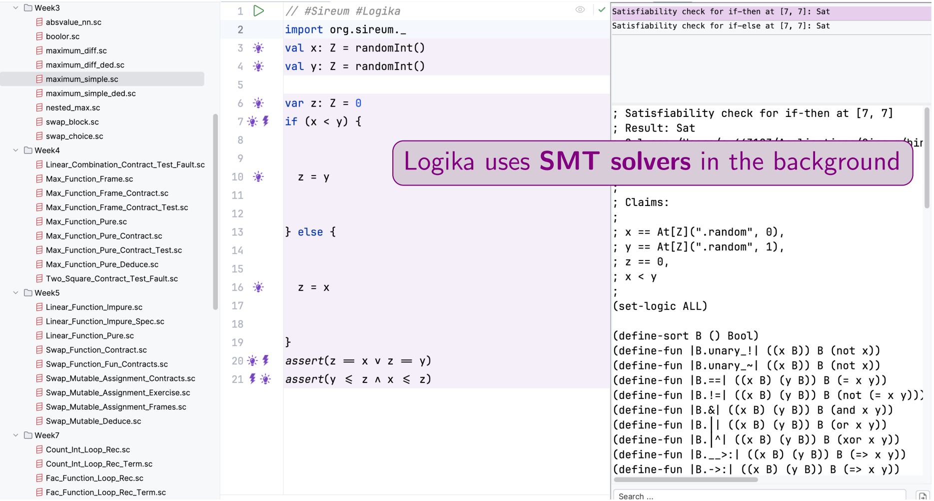
A Quick Tour of Slang and Logika

The User Interface



A Quick Tour of Slang and Logika

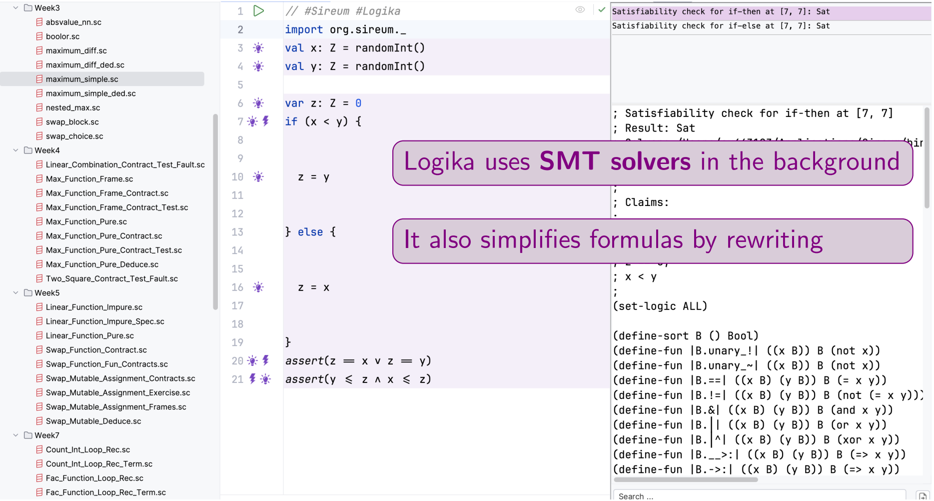
The User Interface



Logika uses **SMT solvers** in the background

A Quick Tour of Slang and Logika

The User Interface

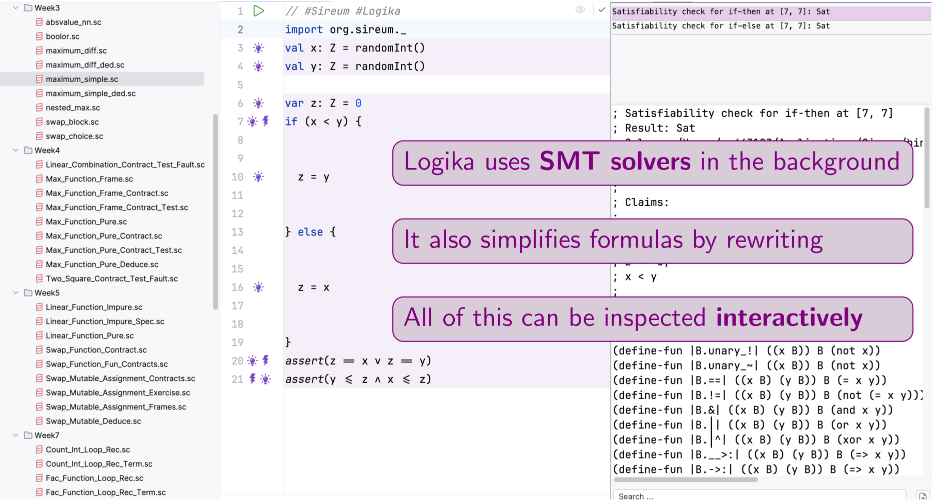


Logika uses **SMT solvers** in the background

It also simplifies formulas by rewriting

A Quick Tour of Slang and Logika

The User Interface



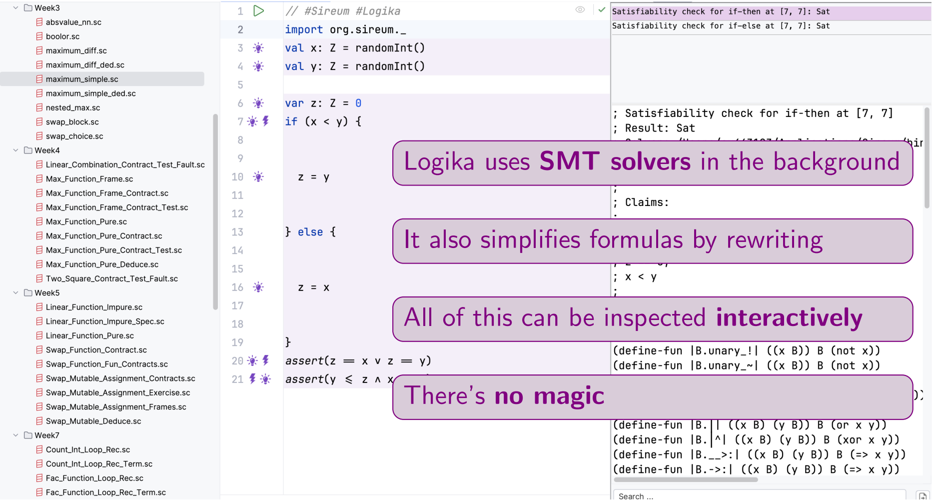
Logika uses **SMT solvers** in the background

It also simplifies formulas by rewriting

All of this can be inspected **interactively**

A Quick Tour of Slang and Logika

The User Interface



Logika uses **SMT solvers** in the background

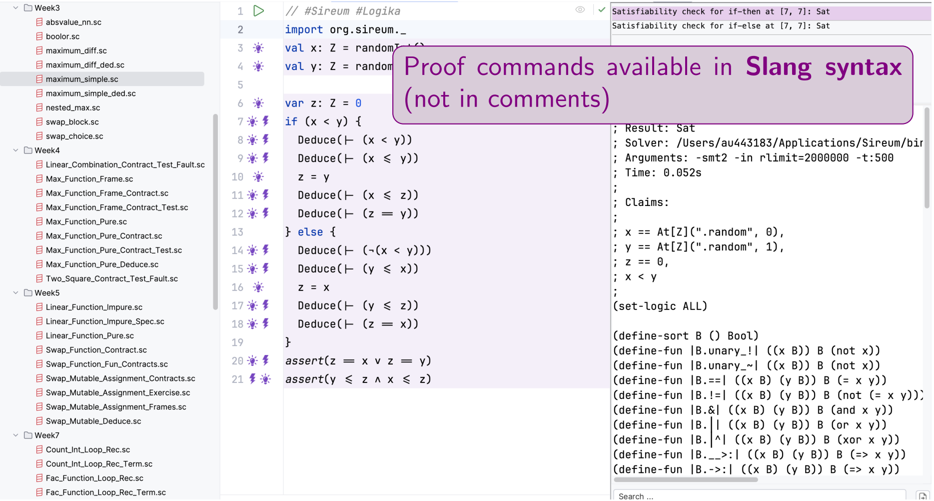
It also simplifies formulas by rewriting

All of this can be inspected **interactively**

There's **no magic**

A Quick Tour of Slang and Logika

The User Interface



Proof commands available in **Slang syntax** (not in comments)

A Quick Tour of Slang and Logika

Contracts & Proof

```
// #Sireum #Logika
import org.sireum._

def swap(a: ZS, i:Z, j: Z) : Unit = {
  Contract(
    Requires(0 ≤ i, i < a.size, 0 ≤ j, j < a.size),
    Modifies(a),
    Ensures(
      a(i) ≡ In(a)(j),
      a(j) ≡ In(a)(i),
      ∀(a.indices)(k ⇒ k ≡ i ∨ k ≡ j ∨ a(k) ≡ In(a)(k)),
      a.size ≡ In(a).size
    )
  )
  val t: Z = a(i)
  Deduce(⊢ (t ≡ In(a)(i)))
  a(i) = a(j)
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
  a(j) = t
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(j) ≡ t))
  Deduce(⊢ (a(j) ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
}
```

Contracts for **compositional reasoning**

A Quick Tour of Slang and Logika

Contracts & Proof

```
// #Sireum #Logika
import org.sireum._

def swap(a: ZS, i:Z, j: Z) : Unit = {
  Contract(
    Requires(0 ≤ i, i < a.size, 0 ≤ j, j < a.size),
    Modifies(a),
    Ensures(
      a(i) ≡ In(a)(j),
      a(j) ≡ In(a)(i),
      ∀(a.indices)(k ⇒ k ≡ i ∨ k ≡ j ∨ a(k) ≡ In(a)(k)),
      a.size ≡ In(a).size
    )
  )
  val t: Z = a(i)
  Deduce(⊢ (t ≡ In(a)(i)))
  a(i) = a(j)
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
  a(j) = t
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(j) ≡ t))
  Deduce(⊢ (a(j) ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
}
```

Contracts for **compositional reasoning**

Hoare-style reasoning about imperative commands

A Quick Tour of Slang and Logika

Contracts & Proof

```
// #Sireum #Logika
import org.sireum._

def swap(a: ZS, i:Z, j: Z) : Unit = {
  Contract(
    Requires(0 ≤ i, i < a.size, 0 ≤ j, j < a.size),
    Modifies(a),
    Ensures(
      a(i) ≡ In(a)(j),
      a(j) ≡ In(a)(i),
      ∀(a.indices)(k ⇒ k ≡ i ∨ k ≡ j ∨ a(k) ≡ In(a)(k)),
      a.size ≡ In(a).size
    )
  )
  val t: Z = a(i)
  Deduce(⊢ (t ≡ In(a)(i)))
  a(i) = a(j)
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
  a(j) = t
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(j) ≡ t))
  Deduce(⊢ (a(j) ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
}
```

Contracts for **compositional reasoning**

Hoare-style reasoning about imperative commands

Proof in Slang as close as possible to **programming**

A Quick Tour of Slang and Logika

Contracts & Proof

Familiar Interface for students

```
// #Sireum #Logika
import org.sireum._

def swap(a: ZS, i:Z, j: Z) : Unit = {
  Contract(
    Requires(0 ≤ i, i < a.size, 0 ≤ j, j < a.size),
    Modifies(a),
    Ensures(
      a(i) ≡ In(a)(j),
      a(j) ≡ In(a)(i),
      ∀(a.indices)(k ⇒ k ≡ i ∨ k ≡ j ∨ a(k) ≡ In(a)(k)),
      a.size ≡ In(a).size
    )
  )
  val t: Z = a(i)
  Deduce(⊢ (t ≡ In(a)(i)))
  a(i) = a(j)
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
  a(j) = t
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(j) ≡ t))
  Deduce(⊢ (a(j) ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
}
```

k: Z

Example

A Quick Tour of Slang and Logika

Contracts & Proof

```
// #Sireum #Logika
import org.sireum._

def swap(a: ZS, i:Z, j: Z) : Unit = {
  Contract(
    Requires(0 ≤ i, i < a.size, 0 ≤ j, j < a.size),
    Modifies(a),
    Ensures(
      a(i) ≡ In(a)(j),
      a(j) ≡ In(a)(i),
      ∀(a.indices)(k ⇒ k ≡ i ∨ k ≡ j ∨ a(k) ≡ In(a)(k)),
      a.size ≡ In(a).size
    )
  )
  val t: Z = a(i)
  Deduce(⊢ (t ≡ In(a)(i)))
  a(i) = a(j)
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
  a(j) = t
  Deduce(⊢ (t ≡ In(a)(i)))
  Deduce(⊢ (a(j) ≡ t))
  Deduce(⊢ (a(j) ≡ In(a)(i)))
  Deduce(⊢ (a(i) ≡ In(a)(j)))
}
```

k: Z

Example

Familiar Interface for students

Interactive inspection of all elements, including formulas and proof

A Quick Tour of Slang and Logika

Proof Information

```
@pure def sorted(seq: ISZ[Z]): B = {
  Contract(
    Ensures(Res == All(1 until seq.size)(i ⇒ seq(i-1) ≤ seq(i)))
  )
  var res: B = true
  var k: Z = 1
  while (k < seq.size) {
    Invariant(
      Modifies(k, res),
      k ≥ 1,
      k-1 ≥ 0,
      k-1 ≤ seq.size,
      seq.size ≥ 2 || res == true,
      seq.size < 2 v k ≤ seq.size,
      seq.size < 2 || res == All(1 until k)(i ⇒ seq(i-1) ≤ seq(i))
    )
    Deduce(⊢ (seq.size ≥ 2))
    if (seq(k - 1) > seq(k)) {
      res = false
    }
    k = k + 1
  }
  Deduce(⊢ (seq.size ≥ 2 || res == All(1 until seq.size)(i ⇒ seq(i-1) ≤ seq(i))))
  Deduce(⊢ (seq.size < 2 || res == All(1 until seq.size)(i ⇒ seq(i-1) ≤ seq(i))))
  return res
}
```

Proof information available **interactively**

A Quick Tour of Slang and Logika

Proof Information

```
@pure def sorted(seq: ISZ[Z]): B = {
  Contract(
    Ensures(Res ≡ All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i)))
  )
  var res: B = true
  var k: Z = 1
  while (k < seq.size) {
    Invariant(
      Modifies(k, res),
      k ⩾ 1,
      k-1 ⩾ 0,
      k-1 ⩽ seq.size,
      seq.size ⩾ 2 || res ≡ true,
      seq.size < 2 ∨ k ⩽ seq.size,
      seq.size < 2 || res ≡ All(1 until k)(i ⇒ seq(i-1) ⩽ seq(i))
    )
    Deduce(⊢ (seq.size ⩾ 2))
    if (seq(k - 1) > seq(k)) {
      res = false
    }
    k = k + 1
  }
  Deduce(⊢ (seq.size ⩾ 2 || res ≡ All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i))))
  Deduce(⊢ (seq.size < 2 || res ≡ All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i))))
  return res
}
```

Click to show some hints

Proof information available **interactively**

"Click the *light bulb*"

A Quick Tour of Slang and Logika

Proof Information

```
@pure def sorted(seq: ISZ[Z]): B = {
  Contract(
    Ensures(Res ≡ All(1 until seq.size)(i ⇒ seq(i-1) ≤ seq(i)))
  )
  var res: B = true
  var k: Z = 1
  while (k < seq.size) {
    Invariant(
      Modifies(k, res),
      k ≥ 1,
      k-1 ≥ 0,
      k-1 ≤ seq.size,
      seq.size ≥ 2 || res ≡ true,
      seq.size < 2 v k ≤ seq.size,
      seq.size < 2 || res ≡ All(1 until k)(i ⇒ seq(i-1) ≤ seq(i))
    )
    Deduce(⊢ (seq.size ≥ 2))
    if (seq(k - 1) > seq(k)) {
      res = false
    }
    k = k + 1
  }
  Deduce(⊢ (seq.size ≥ 2 || res ≡ All(1 until seq.size)(i ⇒ seq(i-1) ≤ seq(i))))
  Deduce(⊢ (seq.size < 2 || res ≡ All(1 until seq.size)(i ⇒ seq(i-1) ≤ seq(i))))
  return res
}
```

```
{ // State claims at line 23
  At(res, 0) ≡ T;
  At(k, 0) ≡ 1;
  k < seq.size;
  k ≥ 1;
  k - 1 ≥ 0;
  k - 1 ≤ seq.size;
  seq.size ≥ 2;
  seq.size < 2 v
    k ≤ seq.size;
  ¬(seq.size < 2);
  res ≡ ∀(1 until k)(i ⇒ seq(i - 1) ≤ seq(i));
  seq(k - 1) > seq(k)
}
```

Filter claims ...

A Quick Tour of Slang and Logika

Proof Information

```
@pure def sorted(seq: ISZ[Z]): B = {
  Contract(
    Ensures(Res ═ All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i)))
  )
  var res: B = true
  var k: Z = 1
  while (k < seq.size) {
    Invariant(
      Modifies(k, res),
      k ⩾ 1,
      k-1 ⩾ 0,
      k-1 ⩽ seq.size,
      seq.size ⩾ 2 || res ═ true,
      seq.size < 2 v k ⩽ seq.size,
      seq.size < 2 || res ═ All(1 until k)(i ⇒ seq(
    )
    Deduce(⊢ (seq.size ⩾ 2))
    if (seq(k - 1) > seq(k)) {
      res = false
    }
    k = k + 1
  }
  Deduce(⊢ (seq.size ⩾ 2 || res ═ All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i))))
  Deduce(⊢ (seq.size < 2 || res ═ All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i))))
  return res
}
```

```
{ // State claims at line 23
  At(res, 0) ═ T;
  At(k, 0) ═ 1;
  k < seq.size;
  k ⩾ 1;
  k - 1 ⩾ 0;
  k - 1 ⩽ seq.size;
  seq.size ⩾ 2;
  seq.size < 2 v
    k ⩽ seq.size;
  ¬(seq.size < 2);
  res ═ ∀(1 until k)(i ⇒ seq(i - 1) ⩽ seq(i));
  seq(k - 1) > seq(k)
}
```

Filter claims ...

Proof information shown to student
close to the program text

A Quick Tour of Slang and Logika

Proof Information

```
5  @pure def sorted(seq: ISZ[Z]): B = {
6    Contract(
7      Ensures(Res == All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i)))
8    )
9    var res: B = true
10   var k: Z = 1
11   while (k < seq.size) {
12     Invariant(
13       Modifies(k, res),
14       k ⩾ 1,
15       k-1 ⩾ 0,
16       k-1 ⩽ seq.size,
17       seq.size ⩾ 2 || res == true,
18       seq.size < 2 ∨ k ⩽ seq.size,
19       seq.size < 2 || res == All(1 until k)(i ⇒ seq(
20     )
21     Deduce(⊢ (seq.size ⩾ 2))
22     if (seq(k - 1) > seq(k)) {
23       res = false
24     }
25     k = k + 1
26   }
27   Deduce(⊢ (seq.size ⩾ 2 || res == All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i))))
28   Deduce(⊢ (seq.size < 2 || res == All(1 until seq.size)(i ⇒ seq(i-1) ⩽ seq(i))))
29   return res
30 }
31
```

```
{ // State claims at line 23
  At(res, 0) == T;
  At(k, 0) == 1;
  k < seq.size;
  k ⩾ 1;
  k - 1 ⩾ 0;
  k - 1 ⩽ seq.size;
  seq.size ⩾ 2;
  seq.size < 2 ∨
    k ⩽ seq.size;
  ¬(seq.size < 2);
  res == ∀(1 until k)(i ⇒ seq(i - 1) ⩽ seq(i));
  seq(k - 1) > seq(k)
}
```

Filter claims ...

Proof information shown to student **close to the program text**

Easy to match program text to formulas (also large formulas)

A Quick Tour of Slang and Logika

Informal vs Formal

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt();
val n: Z = randomInt()
val z: Z = m + n
// deduce z == m + n  (consequence of assignment)
val y: Z = z - n
// deduce z == m + n  (old fact)
// deduce y == z - n  (consequence of assignment)
// deduce y == m      (proof by algebra)
//                    (y == z - n
//                       == (m + n) - n
//                       == m)
val x: Z = z - y
// deduce z == m + n  (old fact)
// deduce y == m      (old fact)
// deduce x == z - y  (consequence of assignment)
// deduce x == n      (proof by algebra)
//                    (x == z - y
//                       == (m + n) - m
//                       == n)
assert(x == n & y == m)
```

Informal proofs in comments useable **without tool support**

A Quick Tour of Slang and Logika

Informal vs Formal

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt();
val n: Z = randomInt()
val z: Z = m + n
// deduce z == m + n  (consequence of assignment)
val y: Z = z - n
// deduce z == m + n  (old fact)
// deduce y == z - n  (consequence of assignment)
// deduce y == m      (proof by algebra)
//                    (y == z - n
//                       == (m + n) - n
//                       == m)
val x: Z = z - y
// deduce z == m + n  (old fact)
// deduce y == m      (old fact)
// deduce x == z - y  (consequence of assignment)
// deduce x == n      (proof by algebra)
//                    (x == z - y
//                       == (m + n) - m
//                       == n)
assert(x == n & y == m)
```

Informal proofs in comments useable **without tool support**

Also used on **white board**

A Quick Tour of Slang and Logika

Informal vs Formal

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt();
val n: Z = randomInt()
val z: Z = m + n
// deduce z ≡ m + n  (consequence of assignment)
val y: Z = z - n
// deduce z ≡ m + n  (old fact)
// deduce y ≡ z - n  (consequence of assignment)
// deduce y ≡ m      (proof by algebra)
//                   (y ≡ z - n
//                      ≡ (m + n) - n
//                      ≡ m)
val x: Z = z - y
// deduce z ≡ m + n  (old fact)
// deduce y ≡ m      (old fact)
// deduce x ≡ z - y  (consequence of assignment)
// deduce x ≡ n      (proof by algebra)
//                   (x ≡ z - y
//                      ≡ (m + n) - m
//                      ≡ n)
assert(x ≡ n ∧ y ≡ m)
```

Logika can do **many proofs fully automatic**

Logika Verified
Proof is accepted

A Quick Tour of Slang and Logika

Informal vs Formal

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt();
val n: Z = randomInt()
val z: Z = m + n
// deduce z == m + n  (consequence of assignment)
val y: Z = z - n
// deduce z == m + n  (old fact)
// deduce y == z - n  (consequence of assignment)
// deduce y == m      (proof by algebra)
//                    (y == z - n
//                       == (m + n) - n
//                       == m)
val x: Z = z - y
// deduce z == m + n  (old fact)
// deduce y == m      (old fact)
// deduce x == z - y  (consequence of assignment)
// deduce x == n      (proof by algebra)
//                    (x == z - y
//                       == (m + n) - m
//                       == n)
assert(x == n ∧ y == m)
```

Logika Verified
Proof is accepted

Logika can do **many proofs fully automatic**

Beginning students benefit from this

A Quick Tour of Slang and Logika

Informal vs Formal

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt();
val n: Z = randomInt()
val z: Z = m + n
// deduce z ≡ m + n  (consequence of assignment)
val y: Z = z - n
// deduce z ≡ m + n  (old fact)
// deduce y ≡ z - n  (consequence of assignment)
// deduce y ≡ m      (proof by algebra)
//                   (y ≡ z - n
//                      ≡ (m + n) - n
//                      ≡ m)
val x: Z = z - y
// deduce z ≡ m + n  (old fact)
// deduce y ≡ m      (old fact)
// deduce x ≡ z - y  (consequence of assignment)
// deduce x ≡ n      (proof by algebra)
//                   (x ≡ z - y
//                      ≡ (m + n) - m
//                      ≡ n)
assert(x ≡ n ∧ y ≡ m)
```

Logika can do **many proofs fully automatic**

Beginning students benefit from this

If **they know that it can be proved**,
Logika confirms or refutes their deductions

Logika Verified
Proof is accepted

A Quick Tour of Slang and Logika

Informal vs Formal

```
// #Sireum #Logika
import org.sireum._

val m: Z = randomInt();
val n: Z = randomInt()
val z: Z = m + n
// deduce z == m + n  (consequence of assignment)
val y: Z = z - n
// deduce z == m + n  (old fact)
// deduce y == z - n  (consequence of assignment)
// deduce y == m      (proof by algebra)
//                    (y == z - n
//                       == (m + n) - n
//                       == m)
val x: Z = z - y
// deduce z == m + n  (old fact)
// deduce y == m      (old fact)
// deduce x == z - y  (consequence of assignment)
// deduce x == n      (proof by algebra)
//                    (x == z - y
//                       == (m + n) - m
//                       == n)
assert(x == n ∧ y == m)

```

Logika Verified
Proof is accepted

Logika can do **many proofs fully automatic**

Beginning students benefit from this

If **they know that it can be proved**, Logika confirms or refutes their deductions

The students can use Logika **like a teacher**

Student Feedback

It was nice with a little mini–project to use some of the techniques learned in the course

Student Feedback

It was nice with a little mini–project to use some of the techniques learned in the course

It was really nice to have exercises during the lecture and that [the teacher] walked around to help us if we were struggling with some of the proofs. I really liked that!

Student Feedback

It was nice with a little mini–project to use some of the techniques learned in the course

It was really nice to have exercises during the lecture and that [the teacher] walked around to help us if we were struggling with some of the proofs. I really liked that!

[The teacher] was really good at explaining the subjects and always made sure that the class was understanding the theory

Student Feedback

It was nice with a little mini–project to use some of the techniques learned in the course

It was really nice to have exercises during the lecture and that [the teacher] walked around to help us if we were struggling with some of the proofs. I really liked that!

[The teacher] was really good at explaining the subjects and always made sure that the class was understanding the theory

I am not sure if I am going to use what if have learned

Discussion

- A good **user-friendly tool** that the students are familiar with is **essential**
- Students look for the **benefit** they get out of a course
- They **don't** have a strong background in maths and logics
- It's better if taught material **does not look like formal methods**
- Concerning proof, **in-class attention** by teacher is required
- Using theorem provers directly **did not work well**
- Notation and methodology should be as close to **programming** as possible
- The students rate this course **very high**: 4.4 out of 5
  (but response rate needs to be improved)
- **Despite its title** "Software Correctness" high number of inscriptions
  (20 students)
- Lecture materials for the course are **publicly available**
  (`https://github.com/santoslab/software-correctness-course-materials`)

Much More Material Available From Kansas State University

S 301: Logical Foundations of Programming

earch...

- ropositional Logic Proofs
- redicate Logic Translations
- redicate Logic Proofs
- lathematical Induction
- ntro to Programming Logic
- unctions and Loops
  - 9.1. Functions
  - 9.2. Recursion
  - 9.3. Loops
  - 9.4. Logika Facts
  - 9.5. Summary
- Sequences, Globals, and mination

U Auto

ing Hugo and Hugo Relearn Theme.

rk is licensed under a Creative

Functions and Loops > Loops

A lot of material **available online**

```
val x: Z = Z.read()
val y: Z = Z.read()

var sum: Z = 0
var count: Z = 0

//prove the invariant before the loop begins
Deduce(
    1 (     sum == 0        )  by Premise,          //from the "sum = 0" assignment
    2 (     count == 0      )  by Premise,          //from the "count = 0" assignment
    3 (     sum == count*x  )  by Algebra*(1, 2)    //proved EXACTLY the loop invariant
)

while (count != y) {
    Invariant(
        Modifies(sum, count),
        sum == count * x
    )

    Deduce(
        1 (     sum == count*x      )   by Premise,     //the loop invariant holds
                                                        //at the beginning of an iteration
    )

    sum = sum + x

    Deduce(
        1 (     sum == Old(sum) + x     )  by Premise,        //from "sum = sum + x" assignment
        2 (     Old(sum) == count*x     )  by Premise,        //loop invariant WAS true, but su
        3 (     sum == count*x + x      )  by Algebra*(1,2)   //current knowledge without using
    )

    count = count + 1

    Deduce(
        1 (     count == Old(count)+ 1  )  by Premise,        //from "count = count + 1" assign
        2 (     sum == Old(count)*x + x )  by Premise,        //from previous "sum = count*x + 
                                                              //but count has changed

        3 (     sum == (count-1)*x + x  )  by Algebra*(1,2),
        4 (     sum == count*x - x + x  )  by Algebra*(3),
        5 (     sum == count*x          )  by Algebra*(4)     //loop invariant holds at end of 
    )
}
```

Much More Material Available From Kansas State University

S 301: Logical Foundations of Programming

earch...

ropositional Logic Proofs
redicate Logic Translations
redicate Logic Proofs
Mathematical Induction
ntro to Programming Logic
unctions and Loops
9.1. Functions
9.2. Recursion
9.3. Loops
9.4. Logika Facts
9.5. Summary
Sequences, Globals, and
mination

U Auto

ing Hugo and Hugo Relearn Theme.

rk is licensed under a Creative

Functions and Loops > Loops

```
val x: Z = Z.read()
val y: Z = Z.read()

var sum: Z = 0
var count: Z = 0

//prove the invariant before the loop begins
Deduce(
    1 (     sum == 0         )   by Premise,
    2 (     count == 0       )   by Premise,
    3 (     sum == count*x   )   by Algebra*(1, 2)
)

while (count != y) {
    Invariant(
        Modifies(sum, count),
        sum == count * x
    )

    Deduce(
        1 (     sum == count*x      )    by Premise,     //the loop invariant holds
                                                         //at the beginning of an iteration
    )

    sum = sum + x

    Deduce(
        1 (     sum == Old(sum) + x     )   by Premise,        //from "sum = sum + x" assignment
        2 (     Old(sum) == count*x     )   by Premise,        //loop invariant WAS true, but su
        3 (     sum == count*x + x      )   by Algebra*(1,2)   //current knowledge without using
    )

    count = count + 1

    Deduce(
        1 (     count == Old(count)+ 1  )   by Premise,        //from "count = count + 1" assign
        2 (     sum == Old(count)*x + x )   by Premise,        //from previous "sum = count*x +
                                                               //but count has changed

        3 (     sum == (count-1)*x + x  )   by Algebra*(1,2),
        4 (     sum == count*x - x + x  )   by Algebra*(3),
        5 (     sum == count*x          )   by Algebra*(4)     //loop invariant holds at end of
    )
}
```

A lot of material **available online**

E.g.

https://textbooks.cs.ksu.edu/cis301/

Much More Material Available From Kansas State University

S 301: Logical Foundations of Programming

earch...

U Auto

ing Hugo and Hugo Relearn Theme.

Functions and Loops > Loops

```
val x: Z = Z.read()
val y: Z = Z.read()

var sum: Z = 0
var count: Z = 0

//prove the invariant before the loop begins
Deduce(
    1 (     sum == 0          )  by Premise,
    2 (     count == 0        )  by Premise,
    3 (     sum == count*x    )  by Algebra*(1, 2)
)

while (count != y) {
    Invariant(
        Modifies(sum, count),
        sum == count * x
    )

    Deduce(
        1 (     sum == count*x      )   by Premise,     //the loop invariant holds
                                                        //at the beginning of an iteration
    )

    sum = sum + x

    Deduce(
        1 (     sum == Old(sum) + x     )   by Premise,         //from "sum = sum + x" assignment
        2 (     Old(sum) == count*x     )   by Premise,         //loop invariant WAS true, but su
        3 (     sum == count*x + x      )   by Algebra*(1,2)    //current knowledge without using
    )

    count = count + 1

    Deduce(
        1 (     count == Old(count)+ 1  )   by Premise,         //from "count = count + 1" assign
        2 (     sum == Old(count)*x + x )   by Premise,         //from previous "sum = count*x +
                                                                //but count has changed
        3 (     sum == (count-1)*x + x  )   by Algebra*(1,2),
        4 (     sum == count*x - x + x  )   by Algebra*(3),
        5 (     sum == count*x          )   by Algebra*(4)      //loop invariant holds at end of
    )
}
```

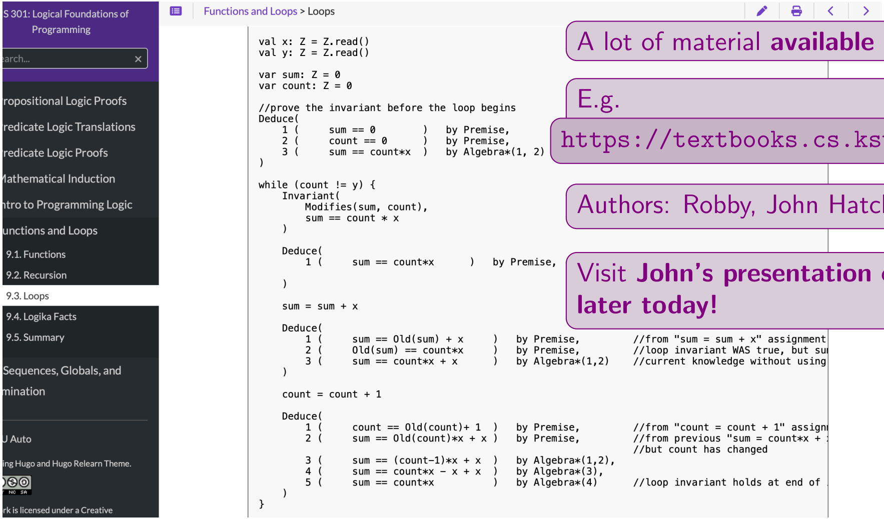
A lot of material **available online**

E.g.

`https://textbooks.cs.ksu.edu/cis301/`

Authors: Robby, John Hatcliff, Julie Thorton

Much More Material Available From Kansas State University



S 301: Logical Foundations of Programming

earch...

ropositional Logic Proofs
redicate Logic Translations
redicate Logic Proofs
Aathematical Induction
ntro to Programming Logic
unctions and Loops
9.1. Functions
9.2. Recursion
9.3. Loops
9.4. Logika Facts
9.5. Summary
Sequences, Globals, and
mination

J Auto

ing Hugo and Hugo Relearn Theme.

rk is licensed under a Creative

Functions and Loops > Loops

```
val x: Z = Z.read()
val y: Z = Z.read()

var sum: Z = 0
var count: Z = 0

//prove the invariant before the loop begins
Deduce(
    1 (     sum == 0          )   by Premise,
    2 (     count == 0        )   by Premise,
    3 (     sum == count*x    )   by Algebra*(1, 2)
)

while (count != y) {
    Invariant(
        Modifies(sum, count),
        sum == count * x
    )

    Deduce(
        1 (     sum == count*x      )   by Premise,
    )

    sum = sum + x

    Deduce(
        1 (     sum == Old(sum) + x     )   by Premise,        //from "sum = sum + x" assignment
        2 (     Old(sum) == count*x     )   by Premise,        //loop invariant WAS true, but su
        3 (     sum == count*x + x      )   by Algebra*(1,2)   //current knowledge without using
    )

    count = count + 1

    Deduce(
        1 (     count == Old(count)+ 1  )   by Premise,        //from "count = count + 1" assign
        2 (     sum == Old(count)*x + x )   by Premise,        //from previous "sum = count*x +
                                                               //but count has changed

        3 (     sum == (count-1)*x + x  )   by Algebra*(1,2),
        4 (     sum == count*x - x + x  )   by Algebra*(3),
        5 (     sum == count*x          )   by Algebra*(4)     //loop invariant holds at end of
    )
}
```

A lot of material **available online**

E.g.

https://textbooks.cs.ksu.edu/cis301/

Authors: Robby, John Hatcliff, Julie Thorton

Visit **John's presentation** on Logika **later today!**

Context, Approach and Evolution

Use and Significance of Slang and Logika

Feedback

Discussion

Next Steps

Next Steps

- Extend the number of **examples**
- Improve support for **self-study**
- improve presentation of **more advanced verification**
- Improve presentation of **proof methodology**
- Rely on discussion and **feedback from students** for improvements
- The course **evolves gradually** – material, tool and students change