Motivations

Programming, Formal Methods, and Secured Systems

Programming

- Emphasize good practices for program development
  - Abstraction, design, specification, testing, verification
- Study new programming techniques
  - Support good program development practices
- Skills to support a secured system
  - Program correctness
  - Dependable software

Formal Methods

Mathematical techniques for the formal specification and development of software/hardware systems; it typically includes:

Formal specification language	Verification tools
• Expresses what the system does • Expresses properties the system should have	• Proof of properties of the specification • Test the properties of the specification

Secured Systems

- Focus on access control, a fundamental topic in secured system
- Study policies and mechanisms that permit or deny the use of a resource or capability
- Examine a logic developed for access control (a formal method) and use it:
  - To formulate access control policies and to describe access control mechanisms
  - As a reasoning tool to help making access control decisions for a secured system

The End

Mathematical Background

Overview

Overview

Prerequisite: discrete mathematics

- Logic
- Sets
- Functions
- Relations

Mathematical Notions and Terminology

Be familiar with the notions and terminology that are used in the following core areas:

- Logic
- Sets
- Functions
- Relations

Definitions, Theorems, and Proofs

- Definitions: write precise mathematical statements to describe the objects and notions we use
- Theorems: correctly interpret theorems (mathematical statements proved true), including the complex ones that comes with lemmas and corollaries
- Proofs: able to verify proofs (logical arguments that demonstrate a statement is true)

Types of Proof

You can identify the types of proof and know the required steps when verifying if the proof is correct; types of proofs includes:

- Proof by construction
  - Example: Show that $p \vee \neg p$ is a tautology by constructing its truth table.
- Proof by contradiction
  - Example: Prove that the square root of 2 is irrational.
- Proof by induction

The End

Mathematical Background

Sets

Basic Set Notation

A **set** is an unordered collection of elements:

- $\{x, y, z\}$ is a set containing three elements.
- $\{\}$ (also written $\emptyset$) is the *empty set*, which contains no elements.
- We write $x \in S$ to indicate x is an element of set S:

$$2 \in \{1, 2, 5\}, \quad w \in \{x, y, z, w\}$$

- We write $x \notin S$ to indicate x is **not** an element of set S:

$$3 \notin \{1, 2, 5\}, \quad 3 \notin \{\}$$

Membership $a \in A$

$\not\subseteq$

Subset relation

$\{ \quad | \quad x \in A \quad \}$

$P(x)$ is true

Set Builder Notation

Subsets

Suppose S and T are both sets.

- S is a *subset* of T, written $S \subseteq T$, provided that every element of S is also an element of T:

$$\{1,5\} \subseteq \{1,2,5\}, \quad \{2\} \subseteq \{1,2,5\}$$

- We write $S \not\subseteq T$ to indicate S is **not** a subset of T:

$$\{1,3,5\} \not\subseteq \{1,2,5\}, \quad \{1,2,5\} \not\subseteq \{2\}$$

- Note that for *every* set S: $\emptyset \subseteq S$, $S \subseteq S$.
- The **power set** of S, written $\mathcal{P}(S)$, is the set containing all subsets of S:

$$\mathcal{P}(\{1,2,5\}) = \{\emptyset, \{1\}, \{2\}, \{5\}, \{1,2\}, \{1,5\}, \{2,5\}, \{1,2,5\}\}$$

Power Set : $\mathcal{P}$

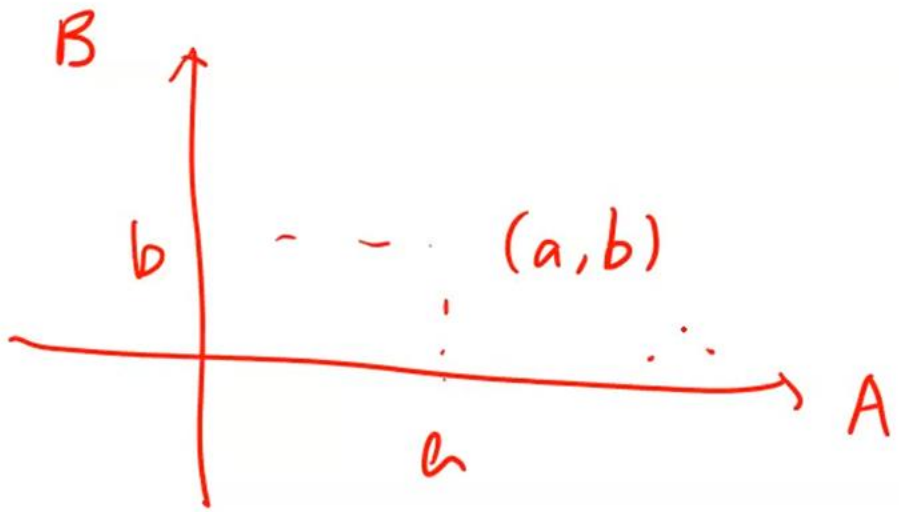
$$\mathcal{P}(\alpha) = \{\alpha\}$$

Basic Set Operations

Notion	Notation	Definition
Union	$S \cup T$	$S \cup T = \{x \mid x \in S \text{ or } x \in T\}$
Intersection	$S \cap T$	$S \cap T = \{x \mid x \in S \text{ and } x \in T\}$
Set Difference	$S - T$	$S - T = \{x \mid x \in S \text{ and } x \notin T\}$
Cartesian Product	$S \times T$	$S \times T = \{(x, y) \mid x \in S \text{ and } y \in T\}$

$$
\begin{aligned}
\{1,2,5\} \cup \{2,4\} &= \{1,2,4,5\} \\
\{1,2,5\} \cap \{2,4\} &= \{2\} \\
\{1,2,5\} - \{2,4\} &= \{1,5\} \\
\{1,2,5\} \times \{2,4\} &= \{(1,2),(1,4),(2,2),(2,4),(5,2),(5,4)\}
\end{aligned}
$$

Cartesian Product $A \times B$



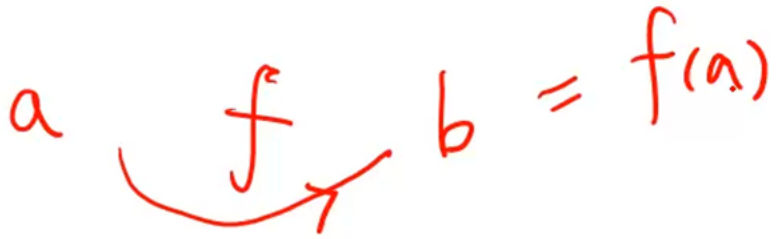
The End

Mathematical Background

Functions

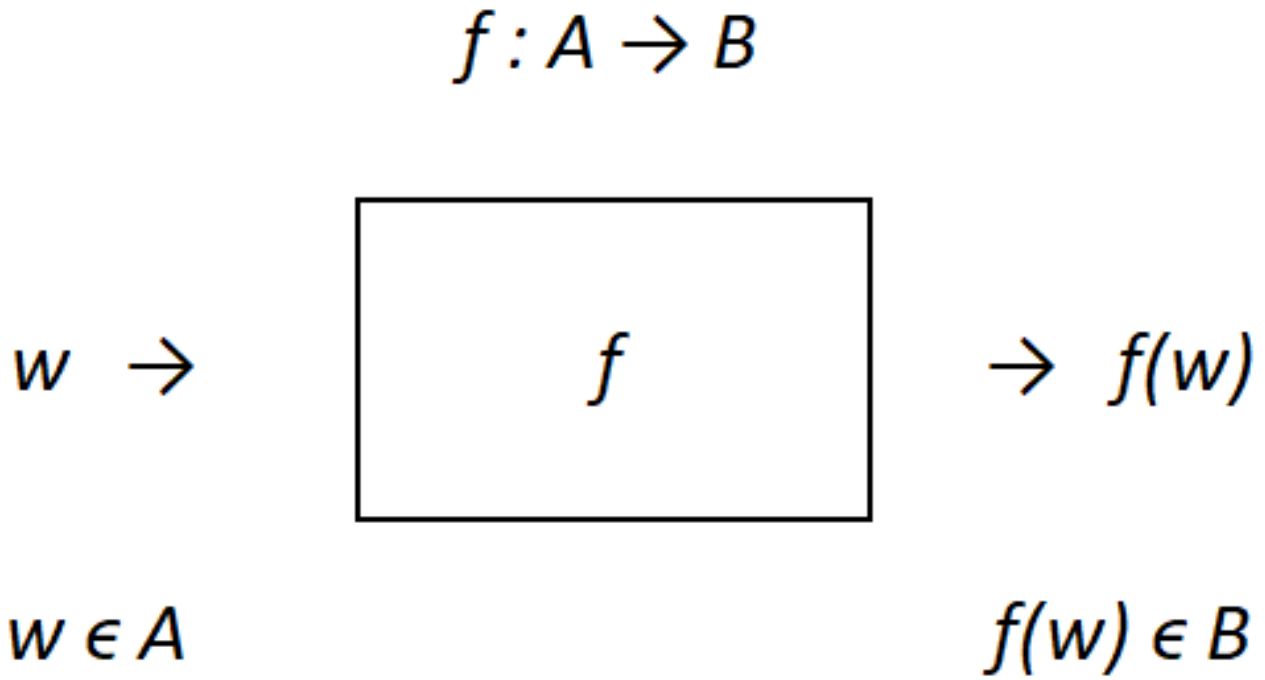
Functions

- The concept of functions is central to mathematics and computer science
- *A function **f** is:*
  - An object to describe an input-output relationship
  - An object where the same input always produces the same output
  - Also called a ***mapping***; we say ***f*** maps ***a*** to ***b*** if ***f(a)*** = ***b***



Functions (cont.)

- ***f*** is a function from a set ***A*** (domain) to a set ***B*** (range):



Function Composition

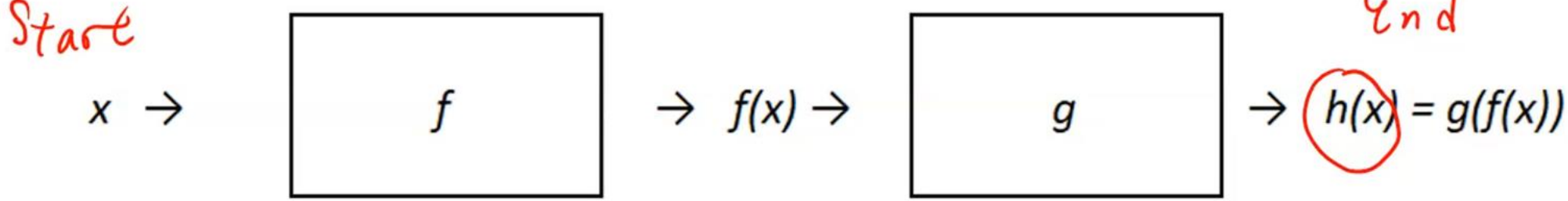
Suppose that $f : A \to B$, $g : B \to C$ are functions.

The ***composition of functions*** f and g is the function $\boldsymbol{h}$ $(\boldsymbol{h} : A \to C)$ *such that* $\boldsymbol{h(x)} = g(f(x))$ *(write* $\boldsymbol{h} = g \circ f$*).*

$h: A \to C$ (write $h = g \circ f$)

$f : A \to B$ $\quad$ $g : B \to C$

Start $\quad$ End



Function Composition (cont.)

Remarks

- A simple way for structuring the design of a program
  - The program is written as a pipeline of operations
  - Easy to see how such designs can be modified
- The composition operator ∘ is:
  - A higher-order function
  - Associative

$f \circ g \circ h$

$(f \circ g) \circ h = f \circ (g \circ h)$

Composition $\circ$:

$$f \circ g(x) = f(g(x))$$

The End

Mathematical Background

Relations

Binary Relations

A **binary relation over a set** A is a set $R \subseteq A \times A$.

- Sample relations over $\{1, 2, 5\}$ include:

$A = \{1, 2, 5\}$

$$\begin{aligned} R_1 &= \{(1,1), (1,2), (2,5), (5,1)\} \\ R_2 &= \{(5,2)\} \\ R_3 &= \{(5,1), (2,2)\} \end{aligned}$$

- $\emptyset$ and $A \times A$ are always relations over A.
- Suppose $a \in A$. The image of relation R under a is $R(a) = \{b \mid (a,b) \in R\}$:

$$R_1(1) = \{1,2\}, \quad R_1(2) = \{5\}, \quad R_1(5) = \{1\}$$

*

Relational Composition

Suppose that R and S are binary relations over A.

- Their **composition** is the relation $R \circ S$ defined by:

$$R \circ S = \{(x, z) \mid \exists y.((x, y) \in R \text{ and } (y, z) \in S)\}.$$

- Recall from previous slide:

$$\begin{aligned} R_1 &= \{(1,1),(1,2),(2,5),(5,1)\} \\ R_3 &= \{(5,1),(2,2)\} \end{aligned}$$

Therefore:

$$\begin{aligned} R_1 \circ R_3 &= \{(1,2),(2,1)\} \\ R_3 \circ R_1 &= \{(5,1),(5,2),(2,5)\} \end{aligned}$$

(1,2)

(1,*) $\in R_1$ $\in R_3$ (*,2)

A Warning on Notation

Parens, curly braces, and square brackets **are not** interchangeable:

$(1,2)$	—	ordered pair $(1, 2)$
$(2,1)$	—	ordered pair $(2, 1)$, which is **not the same** as $(1, 2)$
$\{1, 2\}$	—	two-element set containing 1 and 2
$\{2, 1\}$	—	**same** as $\{1, 2\}$
$\{(1, 2)\}$	—	one-element set containing the pair $(1, 2)$
$[1, 2]$	—	nothing that we'll be talking about right now
$\{\}$	—	the empty set
$\{\{\}\}$	—	a one-element set containing $\{\}$

The End

Functions and Functional Programming

Pure Functions

Pure Functions, Part I

- In procedural programming, functions we develop often come with side effects; for example:
  - An interactive program that reads from the keyboard and writes to the screen as they are running
  - A function that modifies a state variable value/arguments passed and has consequence beyond its scope
- Functions that do not have side effects are referred as pure functions

Pure Functions, Part II

- Pure functions are important in functional programming and formal methods; some benefits are:
  - Simple and precise
  - Easy to test and verify
  - Easy to refactor
- It is easier to reason about programs when they are pure functions

Pure Functions, Part III

- Remarks
  - The Haskell programs that you learn in this class are pure functions
  - While we focus on pure functions in class, one can also use Haskell to write interactive programs (Hutton, Chapter 10)
  - Haskell is also a pure functional language. This means that it has no variable assignments

The End

Programming in Haskell

Introduction

What Is a Functional Language?

Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- Functional programming is a **style** of programming in which the basic method of computation is the application of functions to arguments
- A functional language is one that **supports** and **encourages** the functional style

Example

Summing the integers 1 to 10 in Java:

```
int total = 0;
for (int i = 1; i ≤ 10; i++)
    total = total + i;
```

- The computation method is **variable assignment**

Example (cont.)

Summing the integers 1 to 10 in Haskell:

```
sum [1..10]
```

- The computation method is **function application**

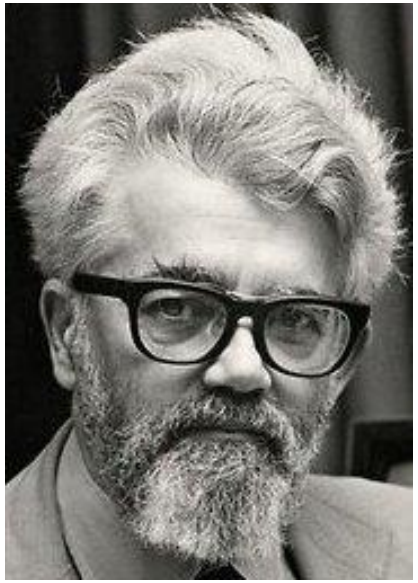
Historical Background, Part I

1930s

- Alonzo Church develops the **lambda calculus**, a simple but powerful theory of functions.

Historical Background, Part II

1950s



- John McCarthy develops **Lisp**, the first functional language, with some influences from the lambda calculus but retaining variable assignments.

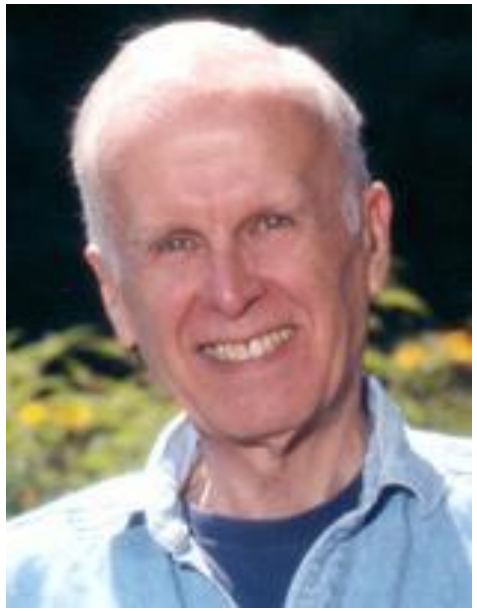
Historical Background, Part III

1960s

- Peter Landin develops **ISWIM**, the first *pure* functional language, based strongly on the lambda calculus, with no assignments.

Historical Background, Part IV

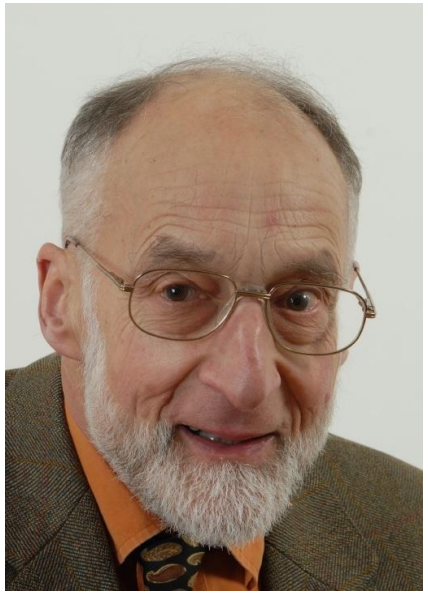
1970s



- John Backus develops **FP,** a functional language that emphasizes *higher-order functions and reasoning about programs.*

Historical Background, Part V

1970s



- Robin Milner and others develop **ML**, the first modern functional language, which introduced *type inference and polymorphic types.*

Historical Background, Part VI

1970s–1980s

- David Turner develops a number of *lazy* functional languages, culminating in the **Miranda** system.

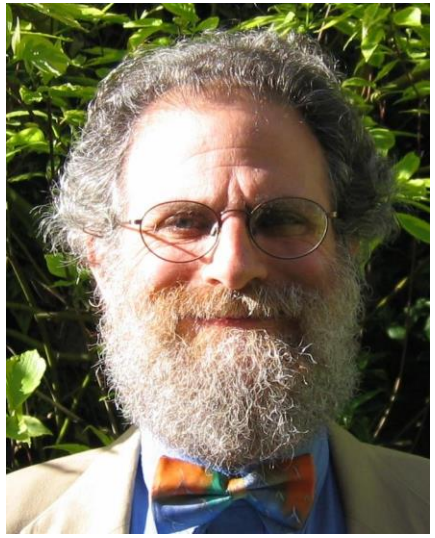
Historical Background, Part VII

1987

- An international committee starts the development of **Haskell**, a standard lazy functional language.

Historical Background, Part VIII

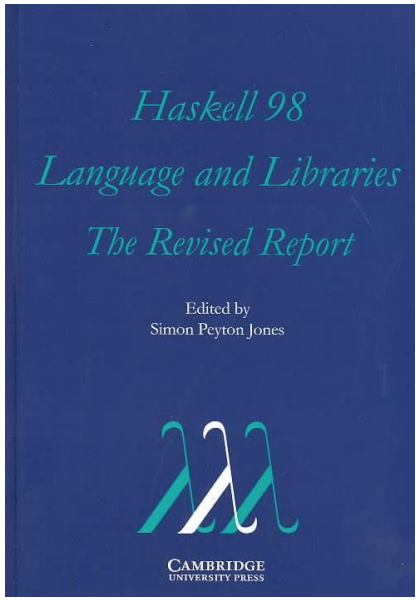
1990s



- Phil Wadler and others develop *type classes* and *monads,* two of the main innovations of Haskell.

Historical Background, Part IX

2003



- The committee publishes the **Haskell Report**, defining a stable version of the language; an updated version was published in 2010.

Historical Background, Part X

2010 to date

- Standard distribution, library support, new language features, development tools, use in industry, influence on other languages, etc.

Historical Background, Part XI

empty list

f.1

```
f []       = []
f (x:xs) = f ys ++ [x] ++ f zs
```

f.2

```
            where
                ys = [a | a ← xs, a ≤ x]
                zs = [b | b ← xs, b > x]
```

?

The End

Functions and Functional Programming

Evaluation of Haskell Expressions

Haskell Expression

- Every well-formed Haskell expression, by definition, has a:
  - Well-formed ***type*** (more discussion coming)
  - ***Value***
- Given a well-typed expression, ***e***, GHCi evaluates ***e*** by reducing it to its simplest form to produce a value and prints it to the terminal (provided ***e*** is printable)

Evaluate Sum [1,2,3]

The sum function (defined by recursion)	Evaluate sum [1,2,3]
sum [] = 0 --- sum.1 ***sum (n:ns) = n + sum ns --- sum.2*** * Base Case Recursive step	sum [1,2,3] (**) = { sum.2 } 1 + sum [2,3] (***) = { sum.2 } 1 + (2 + sum [3]) = { sum.2 } 1 + (2 + (3 + sum [])) = { sum.1 } 1 + (2 + (3 + 0)) = { arithmetic (+ operation)} 6

Feijen's Notation

A proof (calculation) format

- Presents the reasoning using a list of equations
- Identifies the reasons (e.g., the rule used) for each equality in the list

Equation	Interpretation
sum [1,2,3] ***= {sum.2}*** ***1 + sum [2,3]***	Using the rule sum.2, the expression sum [1,2,3] can be ***re-written*** as an equivalent expression: 1+ sum [2,3]

Feijen's Notation (cont.)

Feijen's notation is often used in:

- Equational reasoning (Hutton, Chapter 16.1)
- Reasoning about Haskell programs (Hutton, Chapter 16.2)
- The correctness proofs of Haskell programs

expression 1
= { reason }
expression 2

The End

Access Control, Security, and Trust

Access Control and Computer Security

Chapter 1: Access Control, Security, Trust, and Logic

Access Control, Security, and Trust: A Logical Approach
Shiu-Kai Chin and Susan Older
CRC Press, 2010

Introduction

Intended audience

- Designers, builders, and certifiers of secure and trustworthy computer and information systems

Example concerns

- Who can withdraw funds from a customer's bank account electronically?
- Who is allowed to alter the balance or available funds in a customer's account?
- Who has authority to grant account access?
- What evidence is there to substantiate that the computerized banking system is secure and operating correctly?

Introduction

Intended audience

- Designers, builders, and certifiers of secure and trustworthy computer and information systems

Example concerns

- Who can withdraw funds from a customer's bank account electronically?
- Who is allowed to alter the balance or available funds in a customer's account?
- Who has authority to grant account access?
- What evidence is there to substantiate that the computerized banking system is secure and operating correctly?

Introduction

Intended audience

- Designers, builders, and certifiers of secure and trustworthy computer and information systems

Example concerns

- Who can withdraw funds from a customer's bank account electronically?
- Who is allowed to alter the balance or available funds in a customer's account?
- Who has authority to grant account access?
- What evidence is there to substantiate that the computerized banking system is secure and operating correctly?

Introduction

Intended audience

- Designers, builders, and certifiers of secure and trustworthy computer and information systems

Example concerns

- Who can withdraw funds from a customer's bank account electronically?
- Who is allowed to alter the balance or available funds in a customer's account?
- Who has authority to grant account access?
- What evidence is there to substantiate that the computerized banking system is secure and operating correctly?

Introduction

Intended audience

- Designers, builders, and certifiers of secure and trustworthy computer and information systems

Example concerns

- Who can withdraw funds from a customer's bank account electronically?
- Who is allowed to alter the balance or available funds in a customer's account?
- Who has authority to grant account access?
- What evidence is there to substantiate that the computerized banking system is secure and operating correctly?

Introduction

Intended audience

- Designers, builders, and certifiers of secure and trustworthy computer and information systems

Example concerns

- Who can withdraw funds from a customer's bank account electronically?
- Who is allowed to alter the balance or available funds in a customer's account?
- Who has authority to grant account access?
- What evidence is there to substantiate that the computerized banking system is secure and operating correctly?

Introduction

Intended audience

- Designers, builders, and certifiers of secure and trustworthy computer and information systems

Example concerns

- Who can withdraw funds from a customer's bank account electronically?
- Who is allowed to alter the balance or available funds in a customer's account?
- Who has authority to grant account access?
- What evidence is there to substantiate that the computerized banking system is secure and operating correctly?

What do we mean when we say . . .?

- *Access control* is concerned with the policies and mechanisms that permit or deny the use of a resource or capability.
- *Security* is concerned with the policy and mechanisms that protect the confidentiality, integrity, and availability of information, resources, or capabilities.
- *Trust* focuses on who or what is believed and under what circumstances.

What do we mean when we say . . .?

- *Access control* is concerned with the policies and mechanisms that permit or deny the use of a resource or capability.
- *Security* is concerned with the policy and mechanisms that protect the confidentiality, integrity, and availability of information, resources, or capabilities.
- *Trust* focuses on who or what is believed and under what circumstances.

What do we mean when we say . . .?

- *Access control* is concerned with the policies and mechanisms that permit or deny the use of a resource or capability.
- *Security* is concerned with the policy and mechanisms that protect the confidentiality, integrity, and availability of information, resources, or capabilities.
- *Trust* focuses on who or what is believed and under what circumstances.

The End

Access Control, Security, and Trust

Security and Trust

Trustworthiness

Systems are trustworthy, if:

- they have appropriate policies, mechanisms, and trust assumptions for access control and security
- these policies and mechanisms are logically consistent and correctly implemented

Systems fail due to:

- wear
- flaws
- unintended uses
- wrong operating or design assumptions

Focus is on flaws, unintended uses, and wrong assumptions

Trustworthiness

Systems are trustworthy, if:

- they have appropriate policies, mechanisms, and trust assumptions for access control and security
- these policies and mechanisms are logically consistent and correctly implemented

Systems fail due to:

- wear
- flaws
- unintended uses
- wrong operating or design assumptions

Focus is on flaws, unintended uses, and wrong assumptions

Trustworthiness

Systems are trustworthy, if:

- they have appropriate policies, mechanisms, and trust assumptions for access control and security
- these policies and mechanisms are logically consistent and correctly implemented

Systems fail due to:

- wear
- flaws
- unintended uses
- wrong operating or design assumptions

Focus is on flaws, unintended uses, and wrong assumptions

Trustworthiness

Systems are trustworthy, if:

- they have appropriate policies, mechanisms, and trust assumptions for access control and security
- these policies and mechanisms are logically consistent and correctly implemented

Systems fail due to:

- wear
- flaws
- unintended uses
- wrong operating or design assumptions

Focus is on flaws, unintended uses, and wrong assumptions

Trustworthiness

Systems are trustworthy, if:

- they have appropriate policies, mechanisms, and trust assumptions for access control and security
- these policies and mechanisms are logically consistent and correctly implemented

Systems fail due to:

- wear
- flaws
- unintended uses
- wrong operating or design assumptions

Focus is on flaws, unintended uses, and wrong assumptions

Trustworthiness

Systems are trustworthy, if:

- they have appropriate policies, mechanisms, and trust assumptions for access control and security
- these policies and mechanisms are logically consistent and correctly implemented

Systems fail due to:

- wear
- flaws
- unintended uses
- wrong operating or design assumptions

Focus is on flaws, unintended uses, and wrong assumptions

Trustworthiness

Systems are trustworthy, if:

- they have appropriate policies, mechanisms, and trust assumptions for access control and security
- these policies and mechanisms are logically consistent and correctly implemented

Systems fail due to:

- wear
- flaws
- unintended uses
- wrong operating or design assumptions

Focus is on flaws, unintended uses, and wrong assumptions

Trustworthiness

Systems are trustworthy, if:

- they have appropriate policies, mechanisms, and trust assumptions for access control and security
- these policies and mechanisms are logically consistent and correctly implemented

Systems fail due to:

- wear
- flaws
- unintended uses
- wrong operating or design assumptions

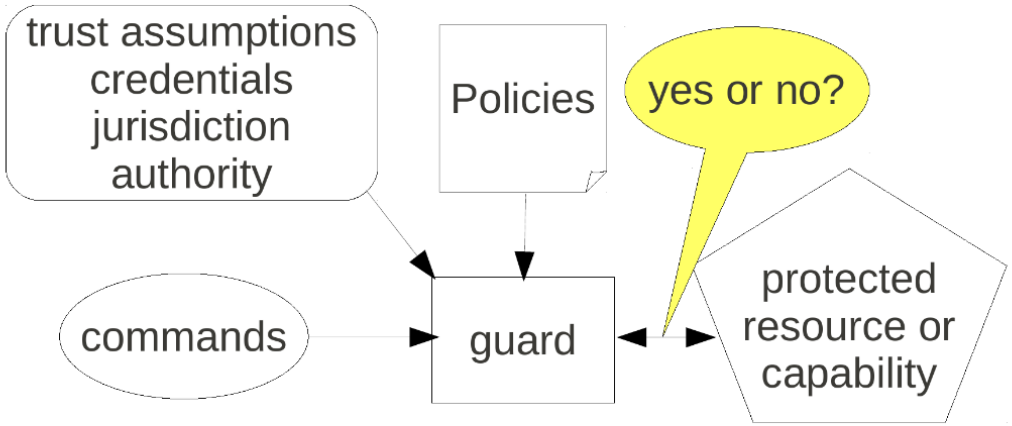
Focus is on flaws, unintended uses, and wrong assumptions

The End

A Logical Approach

Overview

Our Viewpoint

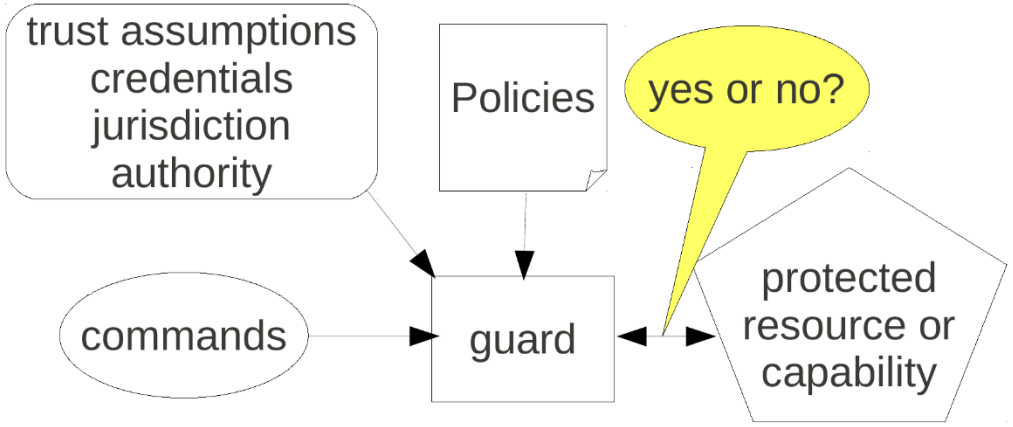


When given a command/request, trust assumptions, credentials, jurisdiction, authority, and policy

- Mathematically justify whether the command/request is honored or not
- Anything less is regarded as a don't know, don't care, or incompetence

No different for hardware designers and verifiers

Our Viewpoint

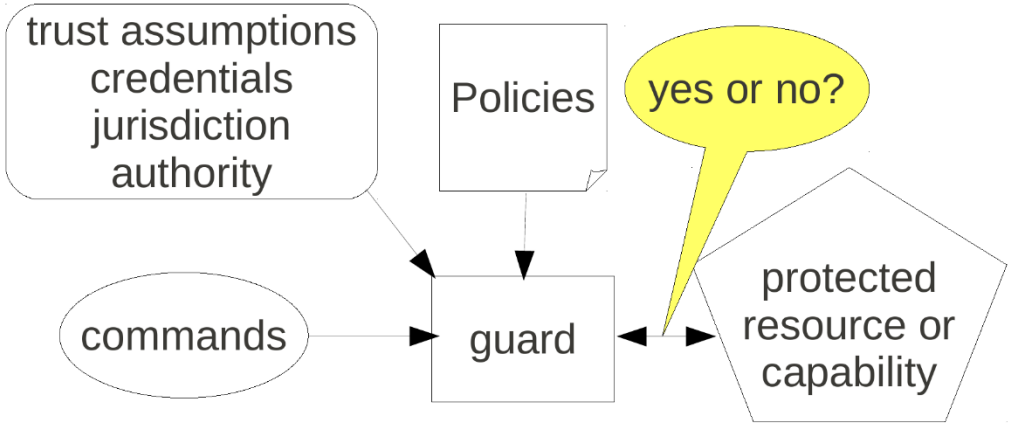


When given a command/request, trust assumptions, credentials, jurisdiction, authority, and policy

- Mathematically justify whether the command/request is honored or not
- Anything less is regarded as a don't know, don't care, or incompetence

No different for hardware designers and verifiers

Our Viewpoint

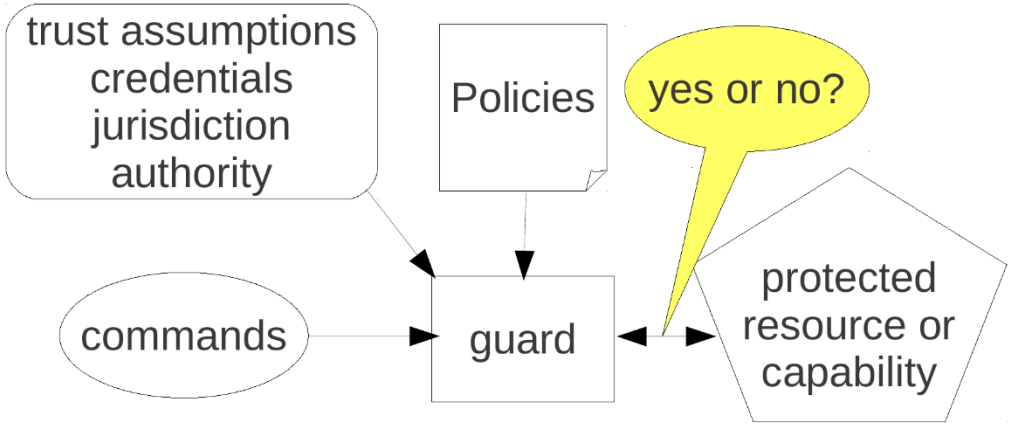


When given a command/request, trust assumptions, credentials, jurisdiction, authority, and policy

- Mathematically justify whether the command/request is honored or not
- Anything less is regarded as a don't know, don't care, or incompetence

No different for hardware designers and verifiers

Our Viewpoint



When given a command/request, trust assumptions, credentials, jurisdiction, authority, and policy

- Mathematically justify whether the command/request is honored or not
- Anything less is regarded as a don't know, don't care, or incompetence

No different for hardware designers and verifiers

A Logical Approach to Access Control

Access-control logic is used in the same way hardware engineers use switching theory and propositional logic to specify, design, and verify hardware

The End

A Logical Approach

First Example (Using Propositional Logic)

Thinking Clearly about Security

Suppose the following two statements are known to be true:

1. If a user requests to see secret data, then the user's authorization is checked.
2. If the user's authorization is checked and the user is authorized at the secret level, then the user is allowed to see secret data.

Is the following statement guaranteed to be true? (*How do you know? And how might you convince your boss?*)

> *If a user requests to see secret data and is allowed to see secret data, then the user is authorized at the secret level.*

Propositional-Logic Representation

Let's introduce the following propositional variables:

$$
\begin{aligned}
RQ &\equiv \text{“User requests to see secret data”} \\
AC &\equiv \text{“User's authorization is checked”} \\
SL &\equiv \text{“User is authorized at secret level”} \\
SD &\equiv \text{“User is allowed to see secret data”}
\end{aligned}
$$

Formulation of the individual statements:

1. $RQ \supset AC$
2. $(AC \wedge SL) \supset SD$
3. $(RQ \wedge SD) \supset SL$

Question: Is the following implication valid?

$$((RQ \supset AC) \wedge ((AC \wedge SL) \supset SD)) \supset ((RQ \wedge \neg SD) \supset \neg SL)$$

Truth Tables for Propositional Formulas

p
T
F

φ	$\neg\varphi$
T	F
F	T

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$
T	T	T
T	F	F
F	T	F
F	F	F

φ_1	φ_2	$\varphi_1 \vee \varphi_2$
T	T	T
T	F	T
F	T	T
F	F	F

φ_1	φ_2	$\varphi_1 \supset \varphi_2$
T	T	T
T	F	F
F	T	T
F	F	T

Sample Truth Table

p	q	r	$q \supset r$	$p \wedge (q \supset r)$	$p \supset q$	$(p \wedge (q \supset r)) \vee (p \supset q)$
T	T	T	T	T	T	T
T	T	F	F	F	T	T
T	F	T	T	T	F	T
T	F	F	T	T	F	T
F	T	T	T	F	T	T
F	T	F	F	F	T	T
F	F	T	T	F	T	T
F	F	F	T	F	T	T

$(p \wedge (q \supset r)) \vee (p \supset q)$ is a tautology (a.k.a. valid), because **all** truth assignments yield a value of T.

In contrast, $p \wedge (q \supset r)$ and $p \supset q$ are **not** tautologies.

Practice

- Translate the following sentences into propositional logic, using the following propositions:

$$H \equiv \text{“ACE leaders are happy with my progress”}$$
$$R \equiv \text{“I ran 8 miles today”}$$
$$S \equiv \text{“I overslept”}$$

1. Either I ran 8 miles today or ACE leaders are not happy with my progress.
2. If I overslept, then I did not run 8 miles today.
3. If I did not oversleep, then ACE leaders are happy with my progress.

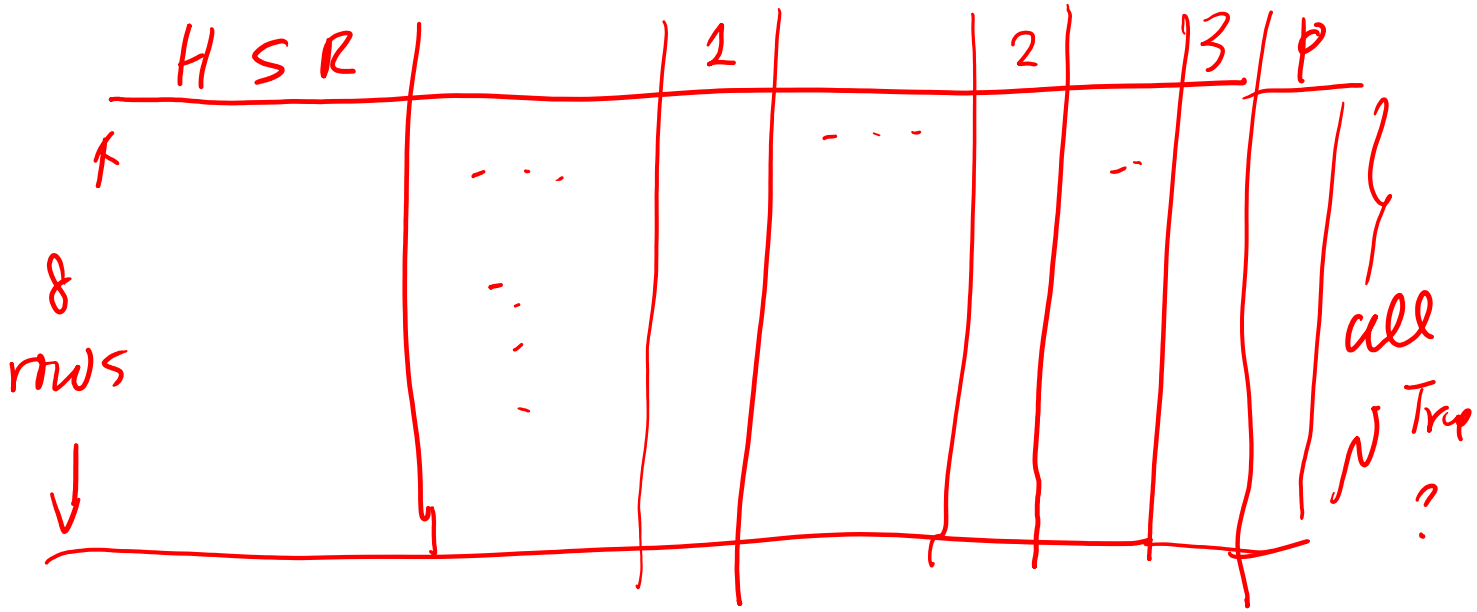
- Use a truth table to determine whether the third statement a logical consequence of the first two (i.e., is $(1 \wedge 2) \supset 3$ valid?).

1. $R \vee H$

2. $S \supset (\neg R)$

3. $(\neg S) \supset H$

Q $(1 \wedge 2) > 3$?



The End

Weekly Summary

Introduction

Programming

Assured programming with a functional language

- Programs are pure functions
- Precise (mathematical) specification of a pure functions and its properties
- The Haskell language

Formal Methods

Mathematical foundations of formal methods

- Formal logic and reasoning
- Mathematical structures
- Sets, relations, and functions
- Discrete structures

Secured Systems: Our Focus

- ***Access control*** is concerned with the policies and mechanisms that permit or deny the use of a resource or capability.
- ***Security*** is concerned with the policy and mechanisms that protect the confidentiality, integrity, and availability of information, resources, or capabilities.
- ***Trust*** focuses on who or what is believed and under what circumstances.

The End