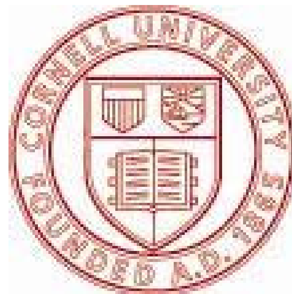
Formal Logical Methods for System Security and Correctness

Logical Foundations of Computer Security

Lecture 1

Robert L. Constable

Cornell University
Department of Computer Science



Marktoberdorf, Germany – Summer 2007

Course Outline

Lecture 1. Basis for trust:
Elements of a Theory of Computing

Lecture 2. Programs, Processes, and Realizers

Lecture 3. A Theory of Events

Lecture 4. Formal Basis for Security Properties

Series Introduction and Type Theory Basics

Motivation: Provide the logical foundations necessary for a trusted information technology, specifically to:

- Formally validate system designs with respect to intent
- Synthesize systems provably faithful to intent
- Explain verified code and safely modify and maintain it
- Protect and secure systems and data

We must be able to express intent formally and then systematically refine the level of detail presented until we can guarantee its correct execution on available hardware.

These lectures will be about formalisms used at the highest levels of abstraction and about their translation to running code. For me the key formalism is type theory, as contrasted with category theory, set theory, algebra, or a high level programming language or informal specification language.

Applied set theory is a closely related approach as in the B-Tool of Abrial. Type theory concepts are programming concepts and thus a basis for synthesis in which the conceptual distance between intent and action is the closest. We exploit this advantage.

Focus of Lecture 1

Security, correctness and trust are related.

To incorporate security services into software, they must first be understood in computing theory, e.g. cryptography.

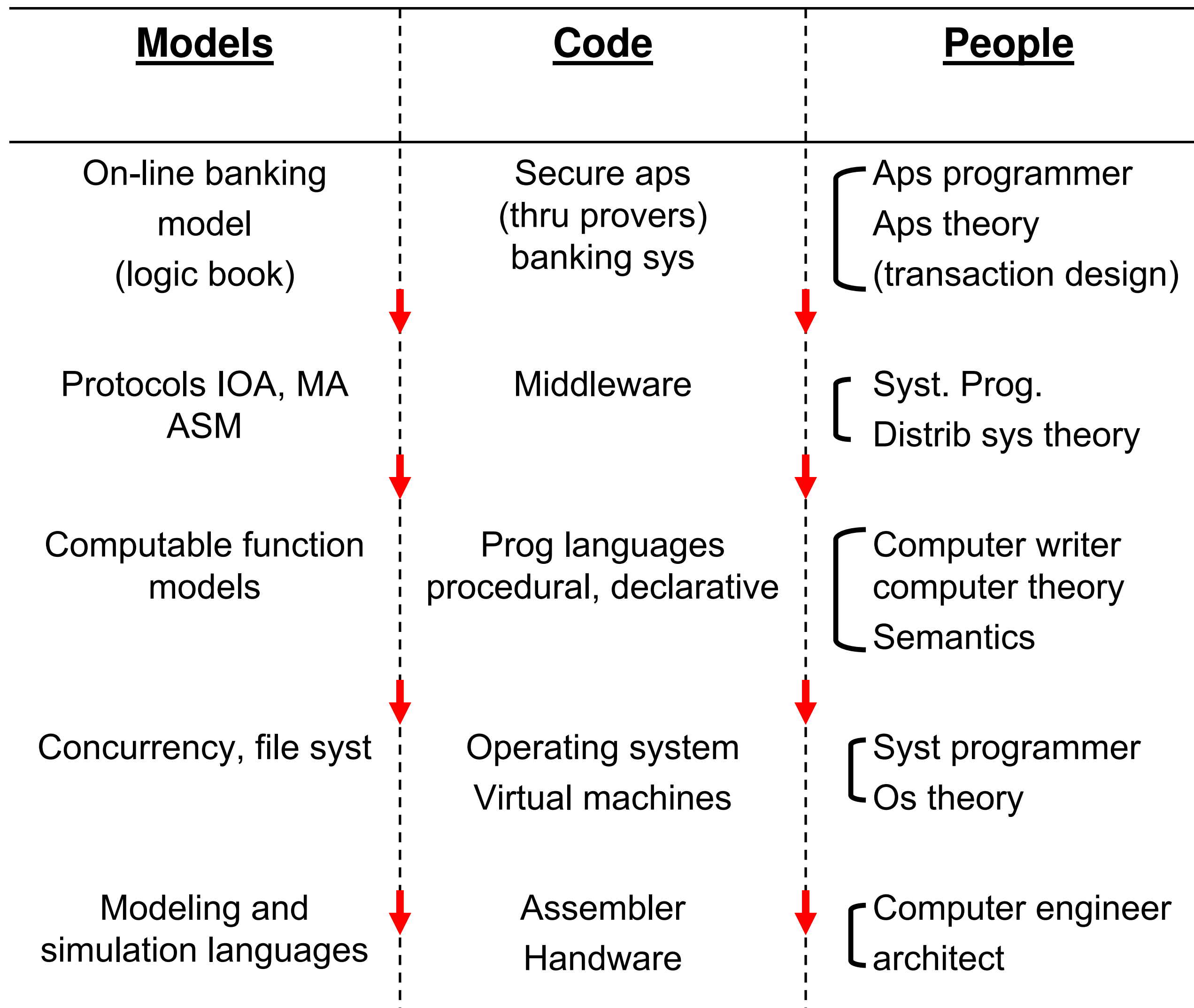
Security services are effective only if elements of software are correct, and correctness must also be part of computer science theory . Computer Science itself is trusted in relation to the larger “organized scientific enterprise,” especially mathematics. So computing theory must connect to the appropriate mathematics, e.g. types to sets.

Focus of Lecture 1 continued

The security mechanism that we introduce in Lecture 4 depends on type theory elements of computing theory. These elements are understood and trusted in relationship to set theory, and to programming language concepts.

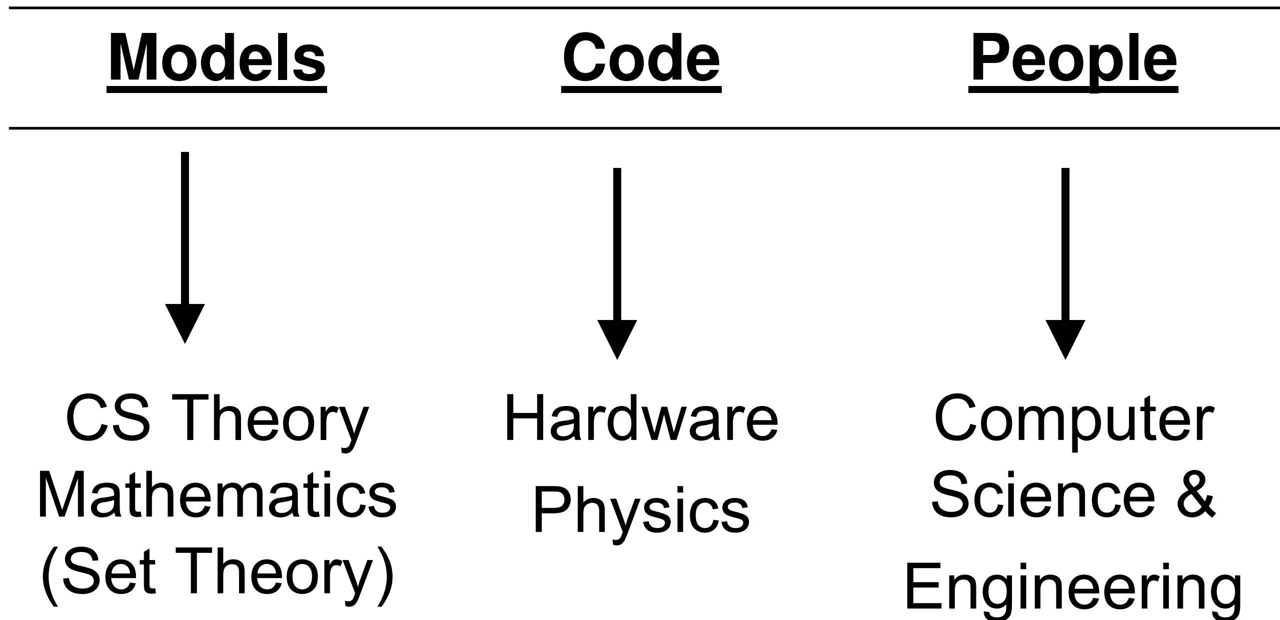
This lecture lays the foundation for the security mechanisms and the distributed computing model in which they are used. Lecture 2 presents the model and Lecture 3 its logic.

Computer Science Practice – “the stack”, translators



This is how we span levels of abstraction

How the stacks are grounded



Refining the Questionaire

What is type?
What is a set?

What is a recursive function?
What is a computable function?
What is a finite automaton?
What is a regular expression?

Can you synthesize an automaton
from a regular expression?
Do you know Java, ML, Lisp, or Scheme?

Historical Perspective

John McCarthy

proposed a theory of computing in 1961 that has motivated many.

Scott	Hoare
Dijkstra	Milner
De Bruijn	Girard
Martin-Löf	

This theory has inspired the MOD series. It in turn was enabled by Alonzo Church's work in the 30's / 40's.

e.g. λ calculus $\rightarrow$ Lisp

Simple Theory of Types $\longrightarrow$ HOL

Historical Perspective

McCarthy said in A Basis for a Mathematical Theory of Computation…

"Computation is sure to become one of the most important of the sciences."

"This is because it is the science of how machines can be made to carryout intellectual processes."

Historical Perspective

McCarthy's Elements of a Theory of Computation

- Expressive notation for computable functions (including partial functions)
- Transforming non-computable specifications to computable ones (an MOD Blue theme)
- Notation for "data spaces" (Hoare)
- Study of equivalence of notations for (partial) functions – recursion induction (Scott extended, Milner implemented)
- Integrating functional and imperative (Algol-like) notations (Hoare, Dijkstra Programming Logics)
- Proof checking and proof generation (Milner onward).

Expressing Tasks Declaratively

- **Sample statement in a Logic of Events**

  Given a network of at least n processes, suppose $m \leq n$ of them are organized into a ring R and requests are sent to elect a leader, then there will exist a unique process declared to be the leader in R.

- **Compare this to**

  Every positive number can be factored in to a product of prime powers, e.g. $1 = 2^0, 6 = 2^1 \cdot 3^1$, etc.

- **Compare to**

  Every planar graph G can be colored by **four** colors.

  The regions of any simple planar map can be colored with only **four** colors in such a way that any two adjacent regions have different colors.

Formalizing Tasks

For formalization, detail matters – too much detail for most people. Consider the simple examples above.

Factoring

$$\forall \boldsymbol{n} : \ \mathbb{N}.^{+} \exists \boldsymbol{F} : \ (\text{PrimePower}) \text{ List}. \ \boldsymbol{prod}(\boldsymbol{F}) = \boldsymbol{n}$$

or

$$\forall \boldsymbol{n} : \{\boldsymbol{i} : \mathbb{N} \mid 1 < \boldsymbol{i}\}.\exists \boldsymbol{F} : (\text{Prime}) \text{ List}. \ \boldsymbol{prod}(\boldsymbol{F}) = \boldsymbol{n}$$

or

$$\forall \boldsymbol{n} : \mathbb{N}.\exists \boldsymbol{F} : (\text{Prime})\text{List}. \ \boldsymbol{prod}(\boldsymbol{F}) = \boldsymbol{n}$$

where $\boldsymbol{prod}(\boldsymbol{nil}) = 1.$

Formalizing Complex Tasks

The detail for the ring of processes example is much more extensive, especially for real code – see **Morrisett**.

How can we digest such detail and connect it to our intuitive thinking? We need computer assistance.

How does formal detail of processes and events relate to detail about imperative programs, functional programs and functions?

These are typical issues faced in my lectures.

Formalization Issues

Consider the formal details about numbers.
John Harrison connects such details to hardware implementations of floating point numbers.
That detail is mainly for specialists.

What about $\mathbb{N}$, the numbers that God gave us, not Intel?

Natural Numbers
$\mathbb{N} = \{0,1,...,10,11,...,36,...100,...1000000,...\}$

What are these?

Numbers

Set Theory: they are sets, elements of the axiomatic inductive set, Inf, i.e.

$\phi \in \boldsymbol{Inf}$ and if $\boldsymbol{x} \in \boldsymbol{Inf}$ then $\boldsymbol{x} \cup \{\boldsymbol{x}\} \in \boldsymbol{Inf}$

Let $\boldsymbol{s}(\boldsymbol{x}) = \boldsymbol{x} \cup \{\boldsymbol{x}\}$, then

$$\phi \in \boldsymbol{Inf} \qquad \frac{\boldsymbol{x} \in \boldsymbol{Inf}}{\boldsymbol{s}(\boldsymbol{x}) \in \boldsymbol{Inf}}$$

Such a set is inductive.

Computing with Numbers

Rules for computing are in a metalanguage

$$x \bullet 0 == 0$$

$$x \bullet (n+1) == (x \bullet n) + x$$

Justified in terms of set theory, but computation is external to set theory.

Numbers

Type Theory: the canonical numbers are elements of the axiomatic inductive type

$0 \in \mathbb{N}$ and if $\boldsymbol{n} \in \mathbb{N}$ then $\boldsymbol{s}(\boldsymbol{n}) \in \mathbb{N}$

or the elements are

$\boldsymbol{nat}\{\boldsymbol{n} : \boldsymbol{number}\}$ $\boldsymbol{n}$ a decimal numeral

the noncanonical numbers are expressions

that reduce to canonical numbers,

e.s. $2+2, 3!, 2^3$, etc.

This is an example of Per Martin-Löf's semantic method.

For numbers in type theory, the theory tells us how to compute. *See Naïve Type Theory section 9.*

Formalization Continued - Functions

The theorem $\forall \boldsymbol{n}:\mathbb{N}.\exists \boldsymbol{F}:(\boldsymbol{Prime})\boldsymbol{List}.\boldsymbol{prod}(\boldsymbol{F})=\boldsymbol{n}$

tells us that there is a function

$\boldsymbol{factor}:\mathbb{N}\rightarrow(\boldsymbol{Prime})\boldsymbol{List}$

$\boldsymbol{factor}(1)=\boldsymbol{nil}$

$\boldsymbol{factor}(2)=\{2\}$

$\boldsymbol{factor}(3)=\{3\}$

$\boldsymbol{factor}(4)=\{2,2\}$

What is a function from A to B?

informal mathematics: a rule of correspondence from A to B

Set Theory:

A functional relation on $\boldsymbol{A} \times \boldsymbol{B}$, i.e. a single-valued set of ordered pairs

$$\boldsymbol{F} \subseteq \boldsymbol{P}(\boldsymbol{A} \times \boldsymbol{B}) \text{ such that}$$

$$\forall \boldsymbol{x} \in \boldsymbol{A}.\ \exists \boldsymbol{y} : \boldsymbol{B}. < \boldsymbol{x}, \boldsymbol{y} > \in \boldsymbol{F}\ \&$$

$$\forall \boldsymbol{p}, \boldsymbol{q} : \boldsymbol{F} \text{ if } \boldsymbol{fst}(\boldsymbol{p}) = \boldsymbol{fst}(\boldsymbol{q})$$

$$\text{then } \boldsymbol{snd}(\boldsymbol{p}) = \boldsymbol{snd}(\boldsymbol{q})$$

Computation is external to ZFC set theory

Formalization of Functions Continued

Type theory: a lambda term $\lambda(x.b)$ such that for all $a \in A, b[a/x] \in B$

The lambda term is the effective rule.

We can apply it to elements of A,

$ap(\lambda(x.b);a)$ reduces to whatever $b[a/x]$ reduces to.

Notice that such a definition is the operational definition in functional programming languages.

Computing in Type Theory

Here is multiplication in type theory,

A lambda term using a fixed point combinator

$$\boldsymbol{fix}(f.\ \lambda(x, y.\ \textbf{if}\ x = 0\ \textbf{then}\ 0\ \textbf{else}\ y + f(x-1, y)\textbf{fi}))$$

Questions

Is $\lambda(\boldsymbol{x}.1) = \lambda(\boldsymbol{x}.\boldsymbol{x} / \boldsymbol{x})$ in $(\mathbb{Q} \rightarrow \mathbb{Q})$?

Is $\lambda(\boldsymbol{x}.\lambda(\boldsymbol{y}.0)) = \lambda(\boldsymbol{y}.0)$?

Formalization – Functions Continued

In Computational Type Theory (CTT) and Intuitionistic Type Theory (ITT) functions are polymorphic. This is a major departure from Set Theory. For example, $\lambda(x.x)$

is the identity function in every function type $A \rightarrow A$,

even if A is empty. In Set Theory there is no such function. The function $hd(L)$ taking the head of a list is polymorphic. We will see that this difference is important in defining data types and subtyping.

Note in CIC, the type theory of Coq, functions, $\lambda x : A \; . \; b$

are monomorphic.

Questions

Does Java have polymorphic functions?

Is 0 polymorphic because it belongs to $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$?

Is $\mathbb{N}$ a subset of $\mathbb{Q}$?

Totalizing Functions

Set theory is easily able to make all functions total.

For example, we can extend integer + to any set.
$n + A = n$ for any set A not an integer.

In type theory it is not possible to make every function total.

We can't tell of an expression A whether it is an integer or not and trying to compute n + *exp* might diverge if exp diverges.

It is very annoying when a set theory formalization uses the "totalizing trick" to make a theory look simple, because that approach does not work constructively.

Formalizations Continued

The factorization theorem used *A* Lists, lists of elements of type *A*. We can imagine theorems using other recursive data structures such as *trees*, *queues*, *stacks*, *streams*, etc.

How are these defined?

Set Theory:

In set theory, the poylmorphic list concept where *A* is any set requires the notion of an inductive class. Typically these are defined as fixed points of monotonic functions over the class of all sets, and the fixed point is guaranteed by a cardinality argument.

$$List(A) = A \cup (A \text{ x } List(A))$$

or

$$List(A) = 1 \oplus (A \text{ x } List(A))$$

These are monotone functions on set, so they have fixed points, e.g. the set

$$A \cup A \text{ x } A \cup A \text{ x } (A \text{ x } A) \cup ...$$

Type Theory:

Recursive types are built by a primitive construction, say

$\mu x.F(x)$ where $F(x)$ is a type expression in x, e.g.

$List(A) = \mu Y.(1 \oplus A \times Y)$.

It suffices for most purposes to use one inductive type, a

W-type (Martin-Löf) to encode recursive ordinals

$$\mu W.(x : A \times (B_x \to W))$$

Formalization

Recall the formulation

$$\forall \boldsymbol{n} : \{\boldsymbol{i} : \boldsymbol{N} \mid 1 < \boldsymbol{i}\}.\exists \boldsymbol{F} : (\mathrm{Pr}\,\boldsymbol{ime})\boldsymbol{List}.\boldsymbol{prod}(\boldsymbol{F}) = \boldsymbol{n}$$

The type $\{\boldsymbol{i} : \mathbb{N} \mid 1 < \boldsymbol{i}\}$ seems like a normal set formed

by separation, but in type theory it also has computational

significance, namely the function

$$\textbf{factor:}\ \{\boldsymbol{i} : \mathbb{N} \mid 1 < \boldsymbol{i}\} \to (\mathrm{Pr}\,\boldsymbol{ime})\boldsymbol{List}$$

does not take evidence for $1<i$ as input

Otherwise, the separation type $\{\boldsymbol{x} : \boldsymbol{A} \mid \boldsymbol{B}\}$ is like the corresponding subset of $\boldsymbol{A}$.

Equality

In Set Theory, equality on the natural numbers reduces to equality of sets. Consider:

$0 = \phi \quad 1=\{\phi\} \quad 2=\{\phi,\{\phi\}\} \quad 3=\{\phi,\{\phi\},\{\phi,\{\phi\}\}\}\ldots$

This is also $0,\{0\},\{0,1\},\{0,1,2\}\ldots$

Notice $\{\phi,\{\phi\}\} = \{\{\phi\},\phi\}$ which is a bit more interesting than 2=2.

In Type Theory, two numbers are equal precisely when they have identical canonical forms. So 2=2 is true and axiomatic. Where 1+1=2 is interesting.

Top Type

Type Theory has collections and objects not found in classical set theory. For example, Top is the type of all closed constructions, considered equal. So it has one element, on the other hand, it has all elements, e.g. $\boldsymbol{A \cap Top = A}$, and

$\boldsymbol{A \sqsubseteq Top}$ for any type A. Top is a signature type for semantic polymorphism a feature exploited by CTT but not by ITT. In CTT, terms with nonterminating reductions are members, such as $\boldsymbol{fix(x.x)}$.

Intersection

In Set Theory $\boldsymbol{A} \cap \boldsymbol{B} = \{\boldsymbol{x} : \boldsymbol{A} \mid \boldsymbol{x} \in \boldsymbol{B}\}$

In Type Theory:

$\boldsymbol{a} = \boldsymbol{b}$ in $\boldsymbol{A} \cap \boldsymbol{B}$ if $\boldsymbol{a} = \boldsymbol{b}$ in $\boldsymbol{A}$ and $\boldsymbol{a} = \boldsymbol{b}$ in $\boldsymbol{B}$

$\{\boldsymbol{x} : \boldsymbol{A} \mid \boldsymbol{P}\} \cap \{\boldsymbol{x} : \boldsymbol{A} \mid \boldsymbol{Q}\} = \{\boldsymbol{x} : \boldsymbol{A} \mid \boldsymbol{P} \,\&\, \boldsymbol{Q}\}$

Notice that $\mathbb{Z}_2 \cap \mathbb{Z}_3 = \mathbb{Z}_6$, e.g. $0 =_2 6$ and $0 =_3 6$

It is easier to compute with $\mathbb{Z}//E_2$ than with $\mathbb{Z}/E_2$

Dependent Intersection

In Type Theory $\boldsymbol{x} : \boldsymbol{A} \cap \boldsymbol{B}_{\boldsymbol{x}}$ is the set of all elements $\boldsymbol{x}$ of $\boldsymbol{A}$ such that $\boldsymbol{x} \in \boldsymbol{B}_{\boldsymbol{x}}$

Subtyping

The type **A** is a subtype of **B** iff

1. Any element of **A** is an element of **B**.
2. If two elements are equal in **A** then they are equal in **B**.

Examples:

$A \sqsubseteq Top$

$Void \sqsubseteq A$

$\{x{:}A \mid P[x]\} \sqsubseteq A$

$A \sqsubseteq A//R$

$\{1,2,3\} \sqsubseteq Z \sqsubseteq Z_6 \sqsubseteq Z_2$

Subtyping relations

If $\mathtt{A} \sqsubseteq \mathtt{A'}$ and $\mathtt{B} \sqsubseteq \mathtt{B'}$ then:

$$
\begin{aligned}
\mathtt{A} + \mathtt{B} &\sqsubseteq \mathtt{A'} + \mathtt{B'} \\
\mathtt{A} \times \mathtt{B} &\sqsubseteq \mathtt{A'} \times \mathtt{B'} \\
\mathtt{A'} \rightarrow \mathtt{B} &\sqsubseteq \mathtt{A} \rightarrow \mathtt{B'} \\
\{\mathtt{x:A} \mid \mathtt{P(x)}\} &\sqsubseteq \mathtt{A} \\
\mathtt{A} \cap \mathtt{B} &\sqsubseteq \mathtt{A} \\
\mathtt{A} \cap \mathtt{B} &\sqsubseteq \mathtt{B} \\
\mathtt{A} &\sqsubseteq \mathtt{A//E} \\
\mathtt{A} &\sqsubseteq \mathtt{A} \cup \mathtt{B} \\
\mathtt{A} &\sqsubseteq \mathtt{Top}
\end{aligned}
$$

Records Naïvely

There are many ways to capture the concept of a record. For example,

```
{x:A; y:B; z:C}
```

can be defined as `A ×(B × C)` and the field selectors can be defined as functions on tuples, say

```
x == λ(r.1of(r))

y == λ(r.1of(2of(r)))

z == λ(r.2of(2of(r)))
```

Naïve Record Extension

We can provide for record extension by adding `Top` as a last component of any record

$$\texttt{Records == T:}U_i \times \texttt{Top}$$

We build up the previous record as follows:

$$R_1 == A \times Top$$

$$R_2 == A \times (B \times Top)$$

$$R_3 == A \times (B \times (C \times Top))$$

Naïve Record Subtyping

Notice that:

$R_2 \sqsubseteq R_1$ **since** $A \sqsubseteq A$, $B \times Top \sqsubseteq Top$

$R_3 \sqsubseteq R_2$ **since** $A \sqsubseteq A$, $B \times (C \times Top) \sqsubseteq B \times Top$

$R_2 \in Records$ **since** $A \in U_i$, $(B \times Top) \in Top$

Records Using Labels

Another approach to records is to take labels, `L`, as indexes into components.

Given	`{x:A; y:B; z:C}`
take	`L={x,y,z}, L` $\sqsubseteq$ `Atom`

Define	`Sig:L` $\rightarrow$ U_i `by`
	`if j=x then A`
	`else if j=y then B else C`

Define the record type as `x:L` $\rightarrow$ `Sig(x).`

Records as Functions

We now take

```
{x:A; y:B; z:C} == x:L → Sig(x).
```

For `r ∈ {x:A; y:B; z:C},`

let `r.i == r(i)`

so `r.x ∈ A, r.y ∈ B, r.z ∈ C`

Records Extension Using Labels

Consider `{x:A; y:B; z:C; w:D}.`

Is this a subrecord of `{x:A; y:B; z:C}?`

To examine this, let `L'={x,y,z,w}.`

Notice `L ⊑ L'.`

Define `Sig'(i)=if i=w then D else Sig(i).`

Notice `x:L'→Sig'(x) ⊑ x:L→Sig(x)`

because `L ⊑ L' and Sig'(x) ⊑ Sig(x) for x ∈ L.`

Record Extension Depends on Function Polymorphism

$$x{:}L' \to Sig'(x) \sqsubseteq x{:}L \to Sig(x)$$

because any function r' in $x{:}L' \to Sig'(x)$ is a function in $x{:}L \to Sig(x)$.

Given inputs from L, x and y, $r'(x) \in Sig'(x)$ and $Sig'(x) = Sig(x)$, $r'(y) \in Sig'(y) = Sig(y)$.

Tension between Types and Computation

Lisp community like to say that Lisp is “typeless.”

More accurately, it is very polymorphic.

Summary of Sets and Types

Sets	Types
ϕ	*void*
Inf	$\mathbb{N}$
$\{x : A \mid P\}$	$\{x : A \mid P\}$
$A \times B$	$A \times B$
$A \cap B$	$A \cap B$
$A \oplus B$	$A + B$
B^A	$A \to B$
$\prod_{x \in A} B_x$	$x : A \to B_x$
$\sum_{x \in A} B_x$	$x : A \times B$
$\mathcal{P}(A)$	$\mathcal{P}_i(A)$
$\mu(x.F)$	$\mu(x.F)$
A / E	$A // E$
$\aleph_k$	$\mathbf{U}_i$
	Top
$A \subset B$	$A \sqsubseteq B$

Reading for Lecture 1

On types

Working Material Chapter 1 Computational Type Theory

2.3 Intuitionistic Type Theory

2.4.1 Subset and Quotient Types

2.5.1 Subtyping

2.5.2 Top

2.5.3 Records

2.6.2 Dependent records

Also Naïve Computational Type Theory

9. Logic and the Peano Axioms

10. Structures, records, and classes

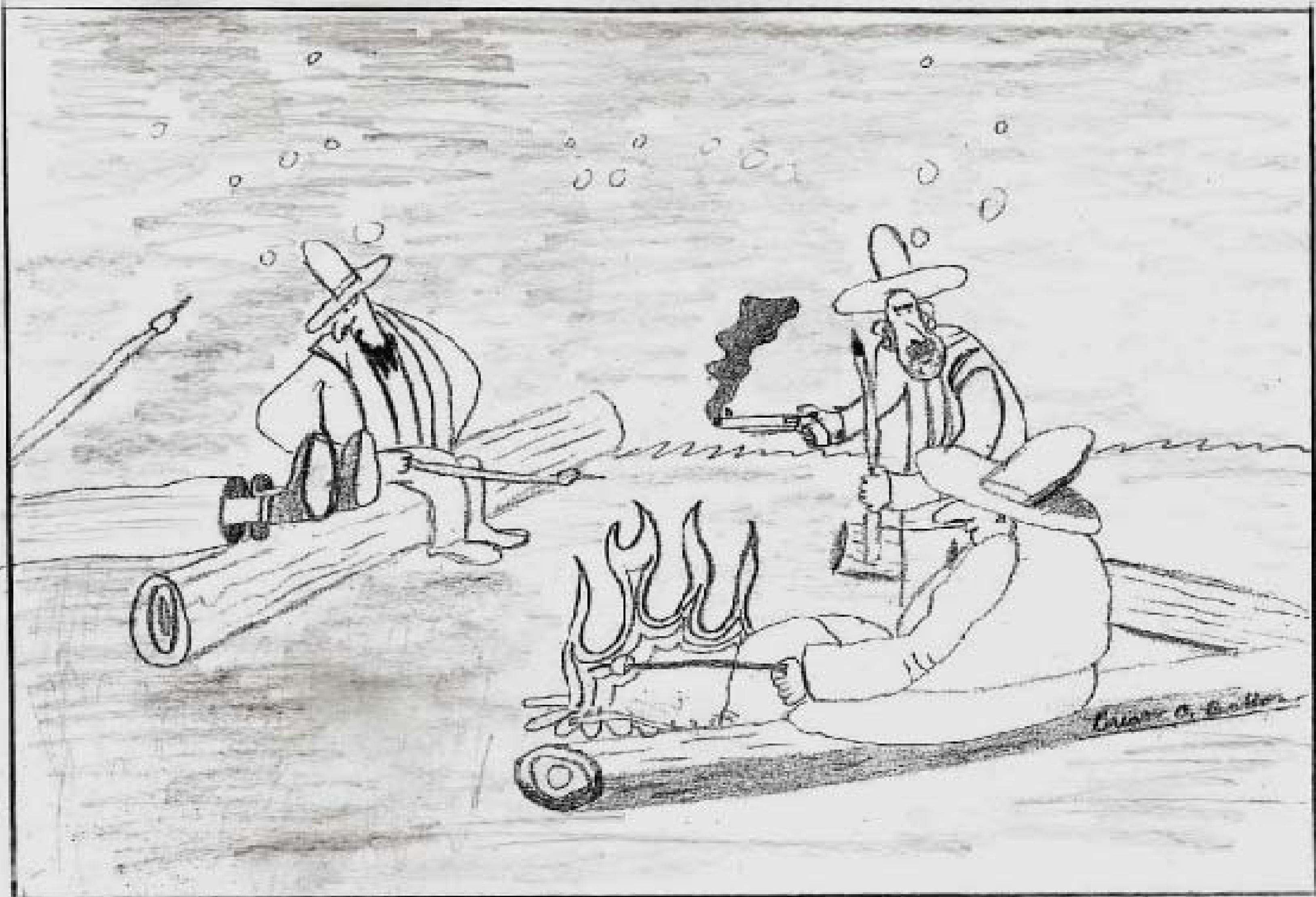
Exercises

- Use AxB, A//E to explain the difference between fractions and rational numbers ($\mathbb{Q}$).
- Can you define $\mathbb{Q}$ so that $\mathbb{Z} \sqsubseteq \mathbb{Q}$?
- Call a type discrete if its equality is decidable.
  - How can we say that $\mathbb{N}, \mathbb{Q}$ are discrete?
  - Is $\mathbb{N} \to \mathbb{N}$ discrete?
- Define: $a @ \ell = \textbf{\textit{if}}$ a =nil $\textbf{\textit{then}}\ \ell$

  $\textbf{\textit{else}}\ hd(a) \bullet (\mathrm{t}\ell(a) @ \ell)\ \textbf{\textit{fi}}$

  Is $\ell_1 @ (\ell_2 @ \ell_3) = (\ell_1 @ \ell_2) @ \ell_3$

  a polymorphic fact or a typed fact?



You guys are both my witnesses... He insinuated that ZFC set theory is superior to Type Theory!

Relationship to Other Lectures

John Mitchell – Protocol Logic, verification and checking tools

Gilles Barthe – Coq certification of security protocols

Helmut Schwichtenberg – Proof with feasible computational content

John Harrison – Interactive and automatic theorem proving

Javier Esparza – Software model checking

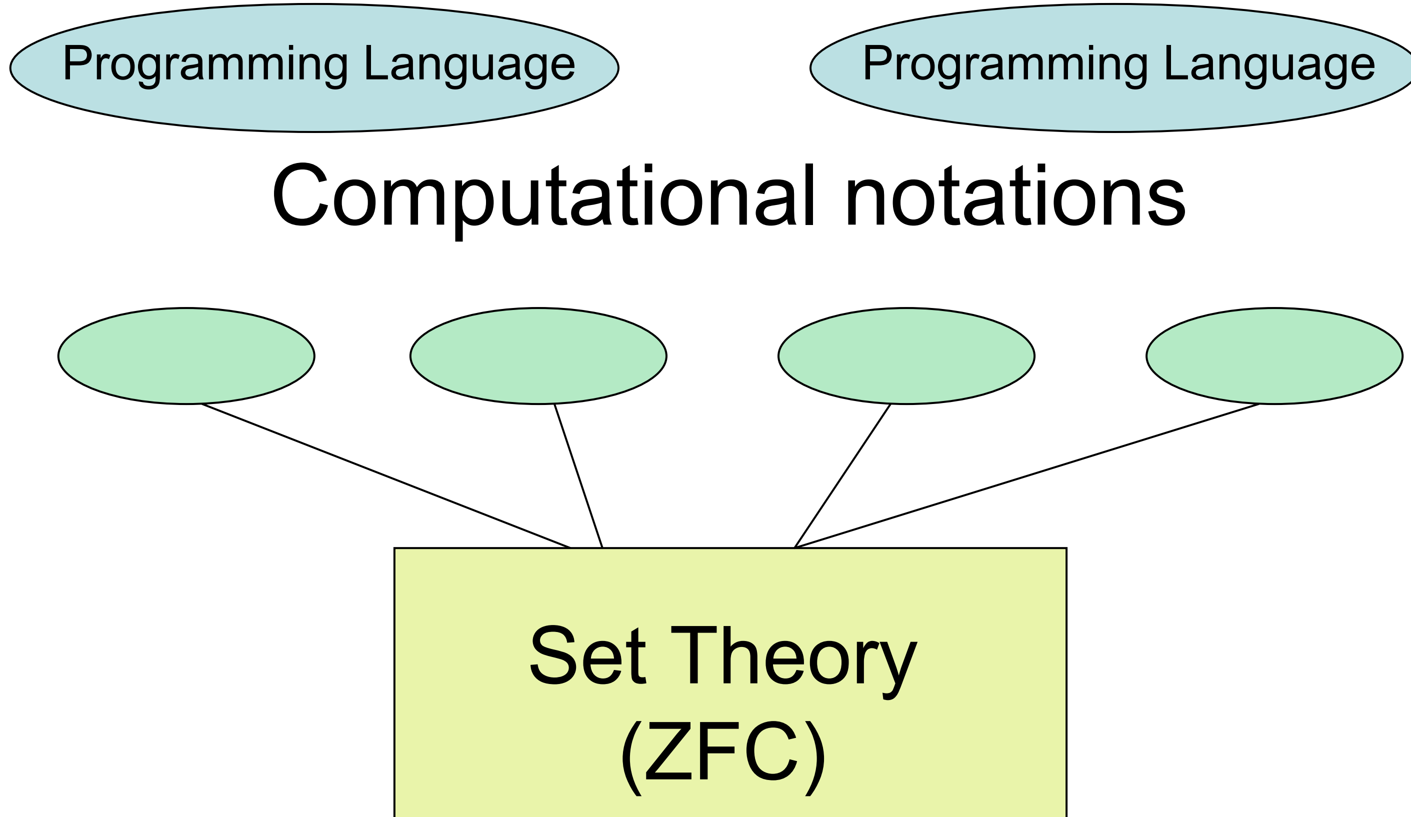
Martin Hyland – Models of recursion and induction

Also ties to **Greg Morrisett, Tobias Nipkow, Martin Hoffman, Orna Grumberg,** and **Stan Wainer**

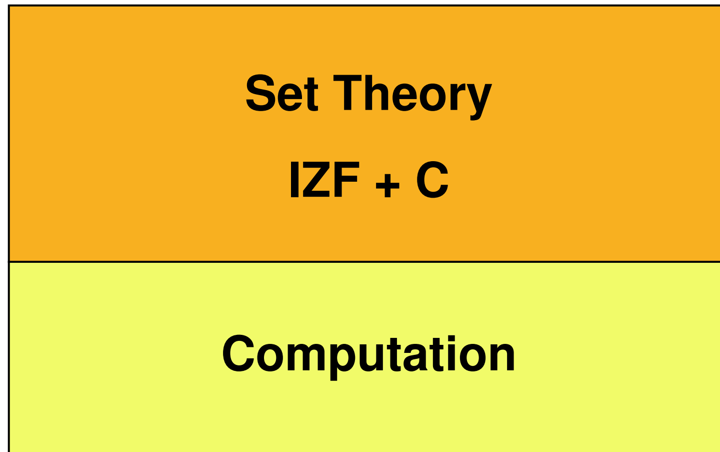
Other connections may unfold.

Standard Methodology

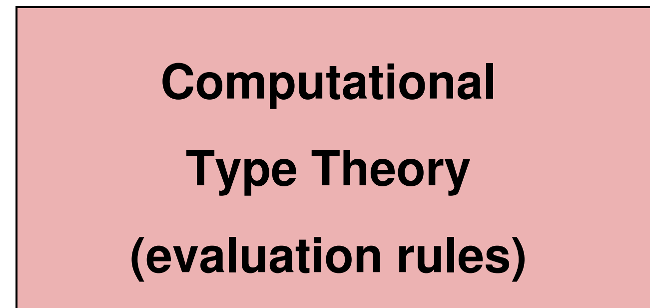
(basis for trusted semantics)



Proposed Modification



or



More About Types

Types rather than sets because of:

- Computability
- Constructive logic
- Subtyping, inheritance
- Openness (extensibility)

Computability requires access to

- Structure
  - Canonical values
  - Computation rules

Abstractness comes from equality.

Formal Logical Methods for System Security and Correctness

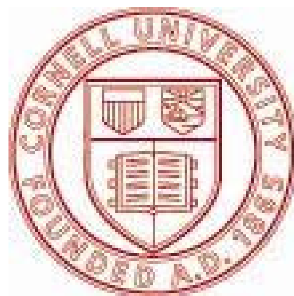
Logical Foundations of Computer Security

Lecture 2

Programs, Processes, and Realizers

Robert L. Constable

Cornell University
Department of Computer Science



Marktoberdorf, Germany – Summer 2007

Programs, Processes, and Realizers

This lecture provides a basis for relating programs and processes to declarative statements.

One relationship starts with a program, say a function

$$f: A \rightarrow B$$

And asserts properties of it, typically

$$\forall x : A.\ R(\mathbf{x}, f(\mathbf{x}))$$

Another relationship starts with a property and finds a function that has it, typically given

$$\forall x : A.\ \exists y : B. R(\mathbf{x}, y)$$

Find the function that produces a value in B satisfying R given a value in A. The function is part of a realizer for the formula

Realizers

When we start with a property or a specification or a goal we say that the function realizes the specification or goal.

Consider: In the logic literature

$$\forall \boldsymbol{x} : \boldsymbol{A}.\boldsymbol{R}(\mathbf{x}, \boldsymbol{f}(\mathbf{x}))$$

$$\forall \boldsymbol{x} : \boldsymbol{A}.\ \exists \boldsymbol{y} : \boldsymbol{B}.\boldsymbol{R}(\mathbf{x}, \boldsymbol{y})$$

Lecture 2 Plan

We will see in detail how to construct realizers from proofs of logical formulas by examining an evidence semantics for formulas. This semantics for constructive logic is known as the Brouwer/Heyting/Kolmogorov (BHK) Semantics, and it is computational. It can be presented in

Computational Type Theory (CTT).

See Naïve Computational Type Theory section 9

MOD 88 Assigning Meaning to Proofs

Lecture 2 - Outline

Semantics of Evidence

Constructive Semantics

Example

Semantics of Proof Objects

Imperative Realizers

Distributed Realizers

Semantics of evidence

Given a formula A, we will define the set or type of objects that count as evidence that A is true in a model $\mathcal{M}$, denote it $[\![A]\!]_{\mathcal{M}}$

Sometimes we suppress the model.

Truth and Evidence:

We expect that there is evidence if A is true in $\mathcal{M}$, thus

$$a \in [\![A]\!]_{\mathcal{M}} \text{ if } \models_{\mathcal{M}} A$$

If A is false, then $[\![A]\!] = \phi$, and if A is true, then $[\![A]\!] \neq \phi$

Propositional Evidence

Suppose that we have evidence sets for the atomic propositions **A,B,C**,…

Here is how to construct evidence for compound formulas:

$$[\![A \,\&\, B]\!] == [\![A]\!] \,\mathrm{x}\, [\![B]\!]$$

$$[\![A \vee B]\!] == [\![A]\!] \oplus [\![B]\!]$$

$$[\![A \Rightarrow B]\!] == [\![A]\!] \rightarrow [\![B]\!]$$

$$[\![\neg A]\!] == [\![A]\!] \rightarrow \phi$$

Evidence for Quantified Statements

$$[\![\exists x : A.B_x]\!] == x : [\![A]\!] \times [\![B_x]\!]$$

$$[\![\forall x : A.B_x]\!] == x : [\![A]\!] \rightarrow [\![B_x]\!]$$

Semantics of Proof Terms

A classical axiom - excluded middle

$$\vdash \boldsymbol{P} \vee \neg \boldsymbol{P} \quad \text{BY} \quad \boldsymbol{Magic}(\boldsymbol{P})$$

$$\boldsymbol{Magic}(\boldsymbol{P}) \in [\![\mathrm{P} \vee \neg \mathrm{P}]\!]$$

Constructive Semantics

If we look only at constructive logic and use types instead of sets, then the semantics of evidence is constructive; it is the Brouwer/Heyting/Kolmogorov semantics.

We recover a classical (Tarski) semantics using

Magic(P) as an oracle.

Formulas and Problems

Here is how we interpret the statements of a typed predicate logic constructively.

For atomic predicates to assert or solve $P(t_1, \ldots, t_n)$ means to provide a proof or a construction p(t1, …, tn)

If P,Q are problem statements (predicate formulas), then to assert

P & Q means to find proofs or constructions p and q for P, Q respectively.

P v Q means to find a proof or construction p for P and mark it as applying to P or to find a proof or construction q for Q and mark it as applying to Q.

Formulas and Problems, continued

$P \Rightarrow Q$	means to find an effective procedure **f** that takes a proof or construction **p** for **P** and computes **p** for **P** and computes **f(p)** a proof or construction for **Q**.
$\neg P$	means that there is no proof or construction for **P**.
$\forall$ x:A.P	means that there is an effective procedure **f** that takes any element of type **A**, say **a**, and computes a proof or construction **f(a)** for **P[a/x].**
$\exists$ x: A.P	means that we can construct an object a of type **A** and find a proof or construction $\mathbf{p_a}$ of **P[a/x],** taken together, $\mathbf{\langle a,p_a \rangle}$ solves this problem or proves this formula.

Programming by refinement

Finding realizers for a goal G can be accomplished by a refinement process.

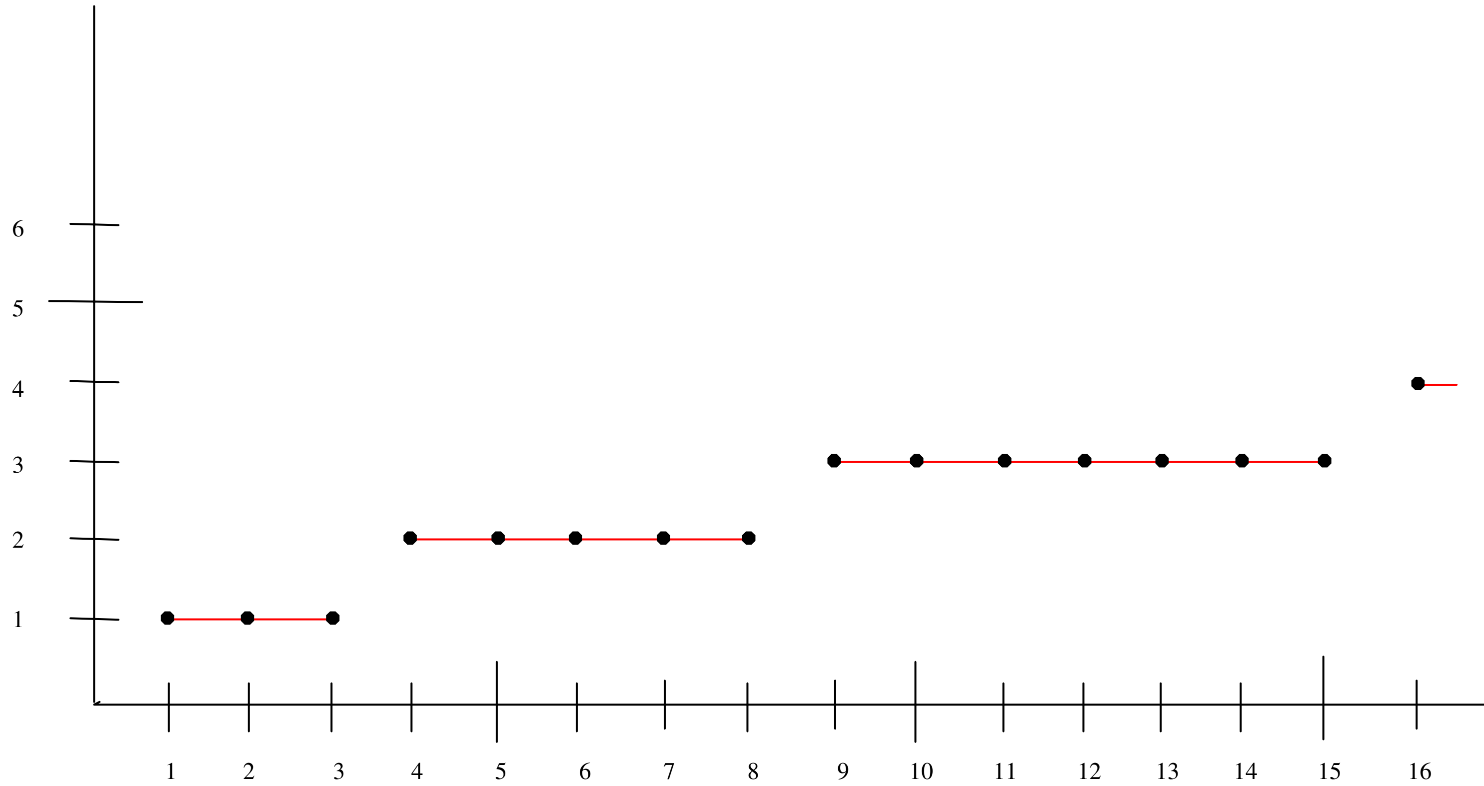
$$\forall \boldsymbol{x} : \boldsymbol{A}.\ \exists \boldsymbol{y} :\ \boldsymbol{B}_1 \mathrm{x}\ \boldsymbol{B}_2.\ \boldsymbol{R}_1(\boldsymbol{x}, \boldsymbol{fst}(\boldsymbol{y}))\ \&\ \boldsymbol{R}_2(\boldsymbol{x}, \boldsymbol{snd}(\boldsymbol{y}))$$

We might decompose this goal into two subgoals

$$\boldsymbol{x} : \boldsymbol{A} \vdash \exists \boldsymbol{y} : \boldsymbol{B}_1 \mathrm{x}\ \boldsymbol{B}_2.\ (\boldsymbol{R}_1\ \&\ \boldsymbol{R}_2)$$

$$\boldsymbol{x} : \boldsymbol{A} \vdash \exists \boldsymbol{y} : \boldsymbol{B}_1 . \boldsymbol{R}_1 \qquad\qquad \boldsymbol{x} : \boldsymbol{A} \vdash \exists \boldsymbol{y} : \boldsymbol{B}_2 . \boldsymbol{R}_2$$

Integer Square Root



Proof of Root Theorem

$\forall n : \mathbb{N}.\ \exists r : \mathbb{N}.\ r^2 \leq n < (r+1)^2$

BY allR

$n : \mathbb{N}$

$\vdash \exists r : \mathbb{N}.\ r^2 \leq n < (r+1)^2$

BY <u>NatInd</u> 1

. . . . .induction case.....

$\vdash \exists r : \mathbb{N}.\ r^2 \leq 0 < (r+1)^2$

BY existsR $\lceil 0 \rceil$ THEN Auto

. . . . .induction case.....

$i : \mathbb{N}^+,\ r : \mathbb{N},\ r^2 \leq i - 1 < (r+1)^2$

$\vdash \exists r : \mathbb{N}.\ r^2 \leq i < (r+1)^2$

BY Decide $\lceil (\underline{r} + 1)^2 \leq i \rceil$ THEN Auto

Proof of Root Theorem (cont.)

. . . . .Case 1.....

$i : \mathbb{N}^{+}, \; r : \mathbb{N}, \; r^2 \leq i - 1 < (r+1)^2, \; (r+1)^2 \leq i$

$\vdash \exists r : \mathbb{N}. \; r^2 \leq i < (r+1)^2$

BY existsR ⌈r + 1⌉ THEN Auto'

. . . . .Case 2.....

$i : \mathbb{N}^{+}, \; r : \mathbb{N}, \; r^2 \leq i - 1 < (r+1)^2, \; \neg\left((r+1)^2 \leq i\right)$

$\vdash \exists r : \mathbb{N}. \; r^2 \leq i < (r+1)^2$

BY existsR ⌈r⌉ THEN Auto

The Root Program Extract

Here is the extract term for this proof in ML notation with proof terms (pf) included:

$$\text{let rec } \mathtt{sqrt}\ i =$$
$$\quad \text{if } i = 0 \text{ then } < 0, \mathtt{pf}_0 >$$
$$\quad\quad \text{else let } < \mathtt{r}, \mathtt{pf}_{i-1} > = \mathtt{sqrt}(i - 1)$$
$$\quad\quad \text{in if } (\mathtt{r} + 1)^2 \le \mathtt{n} \text{ then } < \mathtt{r} + 1, \mathtt{pf}_i >$$
$$\quad\quad\quad \text{else } < \mathtt{r}, \mathtt{pf}_i\text{'} >$$

A Recursive Program for Integer Roots

Here is a very clean functional program

$$
\begin{aligned}
r(n) := \ & \textbf{if } n = 0 \textbf{ then } 0 \\
& \quad \textbf{else} \text{ let } r_0 = r(n-1) \textbf{ in} \\
& \quad \textbf{if } (r_0 + 1)^2 \leq n \textbf{ then } r_0 + 1 \\
& \quad\quad\quad\quad \textbf{else } r_0 \textbf{ fi} \\
& \quad \textbf{fi}
\end{aligned}
$$

This program is close to a declarative mathematical description of roots given by the following theorem.

Theorem $\forall \boldsymbol{n} : \mathbb{N}.\ \exists \boldsymbol{r} : \mathbb{N}.\ \boldsymbol{Root}\ (\boldsymbol{r}, \boldsymbol{n})$

Proof by induction

Base $\boldsymbol{n} = 0$ take $\boldsymbol{r} = 0$, clearly Root $(0,0)$

Induction assume $\exists \boldsymbol{r} : \mathbb{N}.\boldsymbol{Root}\ (\boldsymbol{r},\ \boldsymbol{n}\text{-}1)$

Choose $\boldsymbol{r}_0$ where $\boldsymbol{Root}(\boldsymbol{r}_0, \boldsymbol{n}\text{-}1)$, ***i.e.*** $\boldsymbol{r}_0^{\ 2} \leq \boldsymbol{n}\text{-}1 < (\boldsymbol{r}_0 + 1)^2$

$(\boldsymbol{r}_0 + 1)^2 \leq \boldsymbol{n} \ \vee\ \boldsymbol{n} < (\boldsymbol{r}_0 + 1)^2$

case $(\boldsymbol{r}_0 + 1)^2 \leq \boldsymbol{n}$ then $\boldsymbol{r} = (\boldsymbol{r}_0 + 1)$

$(\boldsymbol{r}_0 + 1)^2 \leq \boldsymbol{n} < ((\boldsymbol{r}_0 + 1)^2 < (\boldsymbol{r}_0 + 2)^2)$

case $\boldsymbol{n} < (\boldsymbol{r}_0 + 1)^2$ then $\boldsymbol{r} = \boldsymbol{r}_0$ since $\boldsymbol{r}_0 \leq \boldsymbol{n}\text{-}1 < \boldsymbol{n}$.

Qed

Deduction Systems

HOL, Nuprl and PVS all use a version of Gentzen's sequents to organize proofs.

$$H_1, \ldots, H_n \vdash G \quad \text{or} \quad H_1, \ldots, H_n \vdash G_1, \ldots, G_m$$

$$\overline{H} \vdash G \qquad\qquad \overline{H} \vdash \overline{G}$$

Typically, hypotheses are named; in Nuprl we use:

$$x_1{:}H_1, \ldots, x_n{:}H_n \vdash G(x_1, \ldots, x_n) \text{ ext } g(x_1, \ldots, x_n)$$

Semantics of Proof Terms - Example

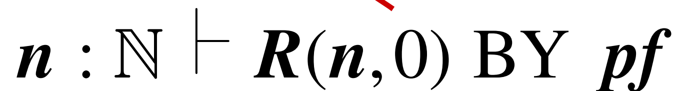
$\boldsymbol{n} : \mathbb{N} \vdash \exists \boldsymbol{r}{:}\mathbb{N}.\, \boldsymbol{R}(\boldsymbol{n},\boldsymbol{r})$ BY $\boldsymbol{existsR}(0, \boldsymbol{pf})$

$\boldsymbol{n} : \mathbb{N} \vdash \boldsymbol{R}(\boldsymbol{n},0)$ BY $\boldsymbol{pf}$

The evidence for $\exists \boldsymbol{r} : \mathbb{N}.\, \boldsymbol{R}(\boldsymbol{n},\boldsymbol{r})$ should be a pair which is what the proof provides $\langle 0, \boldsymbol{pf} \rangle$ written $\boldsymbol{existsR}(0, \boldsymbol{pf})$

Semantics of Proof Terms - Example

$\boldsymbol{n} : \mathbb{N} \vdash \exists \boldsymbol{r}{:}\mathbb{N}.\ \boldsymbol{R}(\boldsymbol{n},\boldsymbol{r})$ BY $\boldsymbol{existsR}(0, \boldsymbol{pf})$



The evidence for $\exists \boldsymbol{r} : \mathbb{N}.\ \boldsymbol{R}(\boldsymbol{n},\boldsymbol{r})$ should be a pair which is what the proof provides $\langle 0, \boldsymbol{pf} \rangle$ written $\boldsymbol{existsR}(0, \boldsymbol{pf})$

Semantics of Proof Terms

Example

$\overline{\mathrm{H}} \vdash \boldsymbol{A} \;\&\; \boldsymbol{B}$ BY and$\boldsymbol{R}\langle \boldsymbol{pfa}, \boldsymbol{pfb} \rangle$

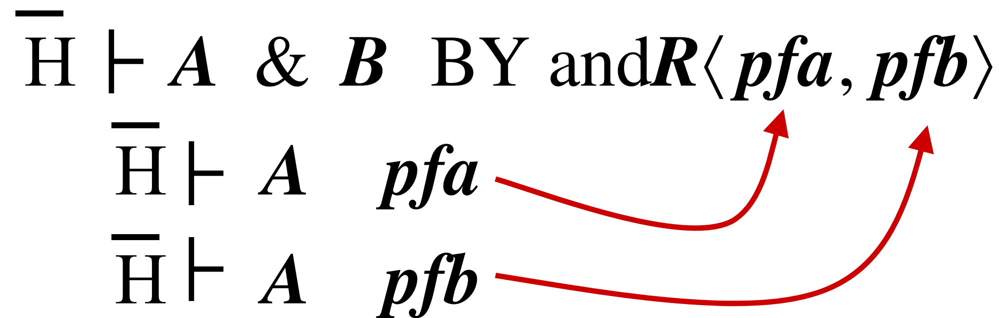
$\overline{\mathrm{H}} \vdash \boldsymbol{A} \quad \boldsymbol{pfa}$

$\overline{\mathrm{H}} \vdash \boldsymbol{A} \quad \boldsymbol{pfb}$

The evidence for $\boldsymbol{A} \;\&\; \boldsymbol{B}$ should be an element of $[\![\boldsymbol{A} \;\&\; \boldsymbol{B}]\!]$ pair, $\langle \boldsymbol{pfa}, \boldsymbol{pfb} \rangle$, the meaning of the proof term and$\boldsymbol{R}(\boldsymbol{pfa}, \boldsymbol{pfb})$.

Semantics of Proof Terms

Example



The evidence for $A \ \& \ B$ should be an element of $[\![A \ \& B]\!]$ pair, $\langle pfa, \ pfb\rangle$, the meaning of the proof term and$R(pfa, \ pfb)$.

Semantics of Proof Terms

$$\overline{\mathrm{H}} \vdash \forall \boldsymbol{x} : \boldsymbol{A}.\ \mathrm{R}(\boldsymbol{x})\ \ \mathrm{BY\ all}\boldsymbol{R}(\boldsymbol{x}.\ \boldsymbol{pfb})$$

$$\overline{\mathrm{H}}, \boldsymbol{x} : \boldsymbol{A} \vdash \boldsymbol{R}(\boldsymbol{x})\ \mathrm{BY}\ \boldsymbol{pfb}$$

Semantics of Proof Terms

$$\overline{\mathrm{H}} \vdash \forall \boldsymbol{x} : \boldsymbol{A}.\ \mathrm{R}(\boldsymbol{x}) \ \text{ BY all } \boldsymbol{R}(\boldsymbol{x}.\ \boldsymbol{pfb})$$

$$\overline{\mathrm{H}},\ \boldsymbol{x} : \boldsymbol{A} \vdash \ \boldsymbol{R}(\boldsymbol{x}) \text{ BY } \boldsymbol{pfb}$$

Constructive Semantics

Notice that the proof term corresponding to

$$\forall x : A.\, R \text{ is } allR(x.\,pf\,)$$

This should denote an element of $x : [\![\mathrm{A}]\!] \rightarrow [\![R_x\,]\!]$

namely $\lambda(x.pf\,)$.

In the constructive case, this function should be computable. We get this result when the evidence sets are types.

Lecture 2 - Outline

Semantics of Evidence

Constructive Semantics

Example

Semantics of Proof Objects

Imperative Realizers

Distributed Realizers

From functional realizers to imperative realizers

Mainstream programming uses state. We need state for distributed computing. Can we extend the realizability interpretation to include state? The simplest way to include state is to model it in the existing theory along the lines shown in **Lecture 1** - use (dependent) records.

An Interative Program for Integer Roots

$r := 0;$

While $(r+1)^2 \leq n$ **do**

$r := r + 1$

od

An Interative Program for Integer Roots, continued

$r := 0;$

While $(r + 1)^2 \leq n$ **do**

$r := r + 1$

od

$\mathbf{n < (r + 1)^2}$

A Program for Integer Roots With Assertions

$r := 0;\ r^2 \leq n$

While $(r+1)^2 \leq n$ **do**

$(r+1)^2 \leq n$

$r := r + 1$

$r^2 \leq n$

od

$r^2 \leq n$

$n < (r+1)^2$

This program suggests a precise specification

Root (r, n) iff $r^2 \leq n < (r+1)^2$

$r^2 \leq n$ is an invariant

While loop realizer

$$\forall s : \{n : \mathbb{N};\ r : \mathbb{N}\}.\ \exists s'' : \{n : \mathbb{N}; r : \mathbb{N}\}.\ Root(s.n,\ s''.r)$$

The proof will build the while loop root finder and apply it to a state s'' where $s''.r = 0$. Thus $s''.r = root(s')$.

Lecture 2 - Outline

Semantics of Evidence

Constructive Semantics

Example

Semantics of Proof Objects

Imperative Realizers

Distributed Realizers

Computations with state: terminating, deterministic

$$S_i : \{n : \mathbb{N}; r : \mathbb{N}\}$$

	n	r
S_0	17	0
S_1	17	1
S_2	17	2
⋮		
S_{17}	17	4

Computations with state: unbounded, deterministic

$$S_i : \{n : \mathbb{N};\, p : \{x : \mathbb{N} \mid Prime(x)\}\}$$

S_0	10	11
S_1	10	13
S_2	10	17
$\vdots$	$\vdots$	

Computations with state: reactive, nondeterministic

in: Input, S: State, out: Output

$$\langle [0], s_0, nil \rangle$$

$$\langle \boldsymbol{nil}, \boldsymbol{s}_1, [15] \rangle$$

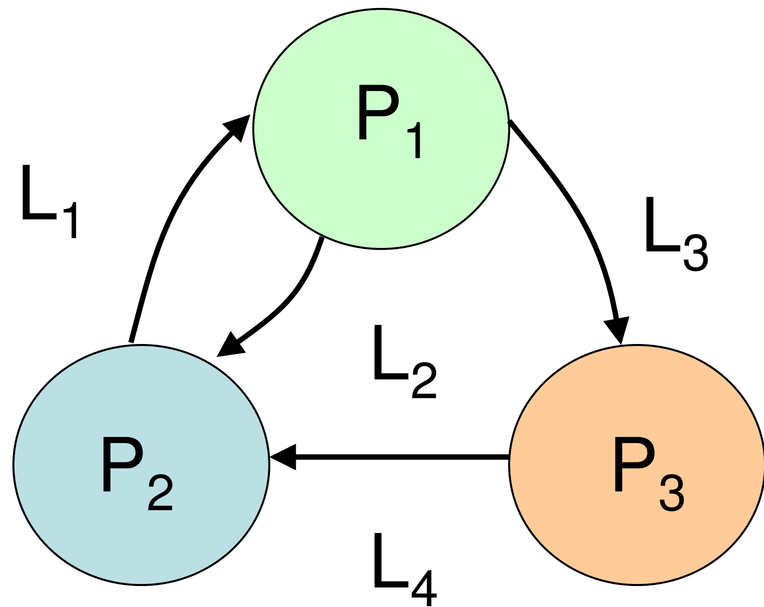
$$\langle \boldsymbol{nil}, \boldsymbol{s}_2, [3] \rangle$$

$$\langle [1,5], \boldsymbol{s}_3, [3,7] \rangle$$

$$\langle [5,8], \boldsymbol{s}_4, \boldsymbol{nil} \rangle$$

$$\vdots$$

Computations with state: asynchronous, distributed



P_i inputs
outputs
state
action

P_i processes

L_i communications channels

Processes are Message Automata

initial

action: receive; effect; send

action: guard; effect; send

frame condition

Message Automata Clauses

- `@i x : T` **initially** `= v`
- `@i` **effect** `k(v : t)` **on** `x`

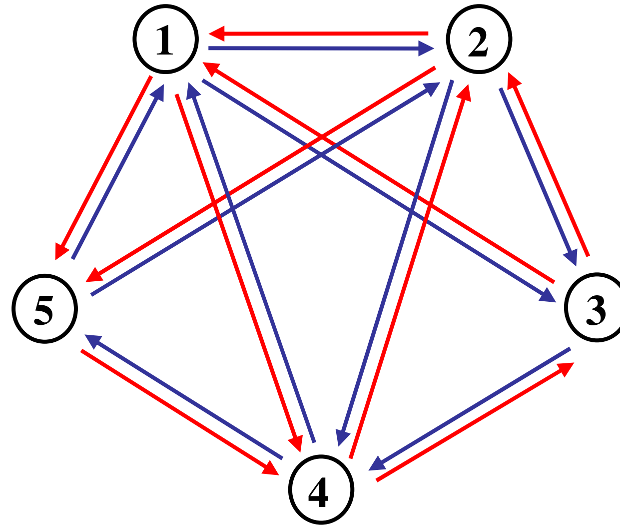
  `x := f state f`
- `@i` **precondition** `a(v:t)` **is** `P state v`
- `k(v : T)` **sends on link l**

  `[tg1, f1 state v; ...; tgn, fn state v]`
- `@i` **only** `[k1, ..., kn]` **sends on link l with tag** `tg`
- `A` $\oplus$ `B`, **where** `A`, `B` **are message automata**

Message Automata – Guessing Roots

```
begin  x, n, r : ℕ, g : ℕ List, rdy : Bool
   initially  n = 2, r = 0, g = nil, rdy = true
      send(< out, < "root ?", n >)
   |*collect guesses*|
   input :receive(< In, < "guess", x >>); g := x•g
   |*check for a root among guesses so far*|
   check :  if rdy=true
        then for all i on g do
             if i² ≤ n<(i + 1)²
                then  r := i
                      rdy := false
                      g := nil
                      send(< out, < "root", < n, r >>>)
                      exit
                fi end
          else skip fi
    |*ask for another root*|
    next: if rdy = false then n := n + 1; rdy := true
                      else skip fi
end
```

Picture of a Computation



loc(1)	loc(2)	loc(3)	loc(4)	loc(5)
$\langle s_1, a_1, m_1 \rangle @ t$	$\langle s_2, a_2, m_2 \rangle @ t$	$\langle s_3, a_3, m_3 \rangle @ t$	$\langle s_4, a_4, m_4 \rangle @ t$	$\langle s_5, a_5, m_5 \rangle @ t$
$\langle s'_1, a'_1, m'_1 \rangle @ t+1$	$\langle s'_2, a'_2, m'_2 \rangle @ t+1$	$\langle s'_3, a'_3, m'_3 \rangle @ t+1$	$\langle s'_4, a'_4, m'_4 \rangle @ t+1$	$\langle s'_5, a'_5, m'_5 \rangle @ t+1$

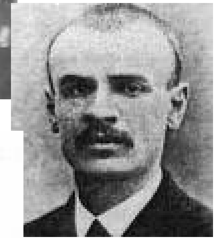
Computational Rules for Distributed realizers

The reduction rules for Message Automata are not fully deterministic because other processes change the communication links, and a scheduler is needed to pick the action.

The semantics assumes a scheduler for each process and allows for an unbounded number of outcomes in one reduction step of a single action.

History

Greeks

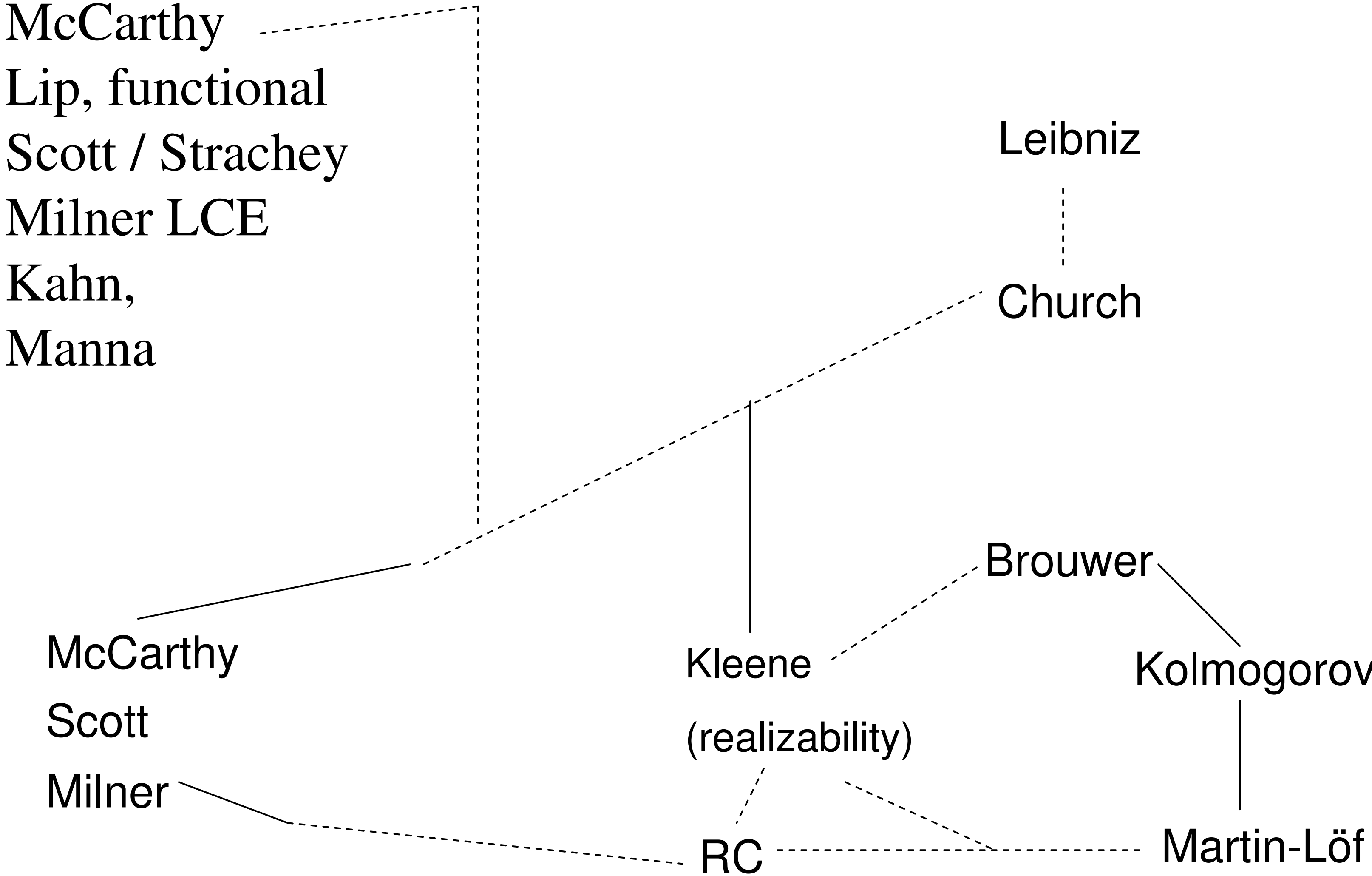


Kronecker
Brouwer
Weyl
Baire
Borel
Bishop

Gentzen
Heyting
Kleene
M-L
de Bruijn

Girard
Coquand

Programming Logics



Reading for Lecture 2

Working Material

Chapter I   Computational Type Theory

2.3.7 Propositions as types

Naïve Computational Type Theory

12 Computational Complexity

Exercise Lecture 2

1. **Write a proof** of:

$$\exists y : B . \forall x : A . R(x, y) \Rightarrow \forall x : A . \exists y : B . R(x, y)$$

   Without using excluded middle, and produce the evidence.

2. **Write a program** that adds assertions to the state

$$\{ n : \mathbb{N} ; r : \mathbb{N} ; a : A ; b : B , c : C \}$$

   such that when the while program halts type *Root(n,r)* has an element, the evidence for its truth

Propositions (cont.)

So `(a=a in A)` is true whenever it is well formed! As a proposition it cannot be denied as long as is sensible.
So `(x=x in A)` is a curiosity as a proposition because it does not make sense to assume it, as in `(x=x in A)` $\Rightarrow$ `P` since to know that this is a sensible proposition is to already know that `(x=x in A)` is true.

So `(x=x in A)` behaves like $x \in A$. It is true as a proposition precisely when $a \in A$ is a correct judgment. (In Nuprl $x \in A$ abbreviates `x=x in A`.

Propositions (cont.)

Type	Proposition
`ax ∈ (a=b in A)`	– we know `a` equals `b` in `A`
`empty (0=1 in N)`	– there is no evidence that `0=1`
`ax ∈ (0≠1 in N)`	– there is evidence that `0` is not equal to `1` in $\mathbb{N}$

It might seem strange that a proposition can also be a type and that a type can also be a proposition. But we will see that all propositions can naturally be construed as types. This is the propositions-as-types principle.

Efficient Root Program

The interactive code and the recursive program are both very inefficient. It is easy to make the recursive program efficient.

$$
\begin{aligned}
&\mathrm{root}(n) := \ \textbf{if}\ n=0\ \textbf{then}\ 0 \\
&\textbf{else}\ \textbf{let}\ r_0 = \mathrm{root}\,(n/4)\ \textbf{in} \\
&\textbf{if}\ (2\cdot r_0+1)^2 \le n \\
&\qquad\qquad \textbf{then}\ 2\cdot r_0+1 \\
&\qquad\qquad \textbf{else}\ 2\cdot r_0\ \textbf{fi} \\
&\qquad\qquad \textbf{fi} \\
&\qquad\qquad \text{since if } n \ne 0,\ n/4 < n
\end{aligned}
$$

This is an efficient recursive function, but why is it correct?

A Theorem that Roots Exist (Can be Found)

Theorem $\forall \boldsymbol{n}: \mathbb{N}.\ \exists \boldsymbol{r}: \mathbb{N}.\ \boldsymbol{Root}\ (\boldsymbol{r}, \boldsymbol{n})$

Pf by efficient induction

Base $\boldsymbol{n} = 0$ let $\boldsymbol{r} = 0$

Induction case assume $\exists \boldsymbol{r}: \mathbb{N}.\boldsymbol{Root}\ (\boldsymbol{r},\ \boldsymbol{n}/4)$

Choose $\boldsymbol{r}_0$ where $\boldsymbol{r}_0^{\ 2} \leq \boldsymbol{n}/4 < (\boldsymbol{r}_0 + 1)^2$

note $4 \cdot \boldsymbol{r}_0^{\ 2} \leq \boldsymbol{n} < 4 \cdot (\boldsymbol{r}_0 + 1)^2 = 4 \cdot \boldsymbol{r}_0^{\ 2} + 8 \cdot \boldsymbol{r}_0 + 4$

thus $2 \cdot \boldsymbol{r}_0 \leq \boldsymbol{root}\ (\boldsymbol{n}) < 2 \cdot (\boldsymbol{r}_0 + 1)$

if $(2 \cdot \boldsymbol{r}_0 + 1)^2 \leq \boldsymbol{n}$ then $\boldsymbol{r} = 2 \times \boldsymbol{r} + 1$

since $(2 \cdot \boldsymbol{r}_0)^2 = 4 \cdot \boldsymbol{r}_0^{\ 2} + 8 \cdot \boldsymbol{r}_0 + 4$

else $\boldsymbol{r} = 2 \times \boldsymbol{r}$ since $(2 \cdot \boldsymbol{r}_0)^2 \leq \boldsymbol{n} < (2 \cdot \boldsymbol{r}_0 + 1)^2$

Qed

Exercise

Show how to create a realizer for Root(n,r)

in:

$r := 0;\ r^2 \leq n$

While $(r+1)^2 \leq n$ **do**

$r := r + 1$

od

Root(n,r)

Lecture 3

A Theory of Events

Robert L. Constable

Marktoberdorf Lecture 3
July 2007

The Challenge

The logic and technology of functional programs are elegant and useful.

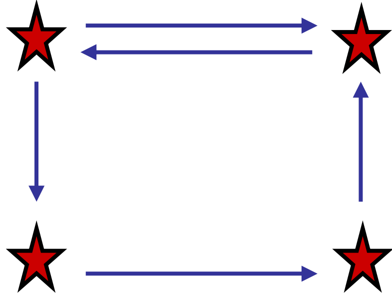
So are high level procedural logics (no pointers), especially asserted programming logics (from the 1970's onward to industrial tools).

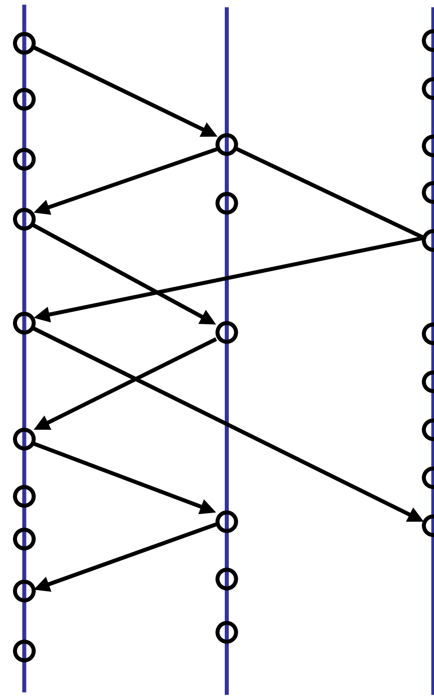
Can we find an elegant theory of distributed processes with process extraction capability? What is the right specification language?

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proofs

Events in Space/Time





Intro to Event Systems

We captured a semantics for processes in the standard asynchronous message passing distributed computing models.

Our design of the **Logic of Events** was strongly informed by the systems group and partners.

We captured the conceptual level at which the protocol designers worked.

event based analysis

high level atomicity

Intro to Event Systems, continued

We then abstracted many high level distributed system concepts into an accessible logic with practical value: many details can be automatically added back by the extractors.

to MA

to Java

Intro to Event Systems, continued

The level of abstraction we achieved, reveals many interpretations, “good science stories,” as well as good “technology stories.”

The general setting is observables in time/space.

Space: locations at which “things happen”

Time: events happening at locations.

References

Leslie Lamport

Time, clocks and the ordering of events in a distributed system, CAM 21, 1978

Glynn Winskel

Events in Computation

PhD Thesis, Univ. of Edinburgh, 1980

Relationship to Winskel, Lamport

As in Leslie Lamport's papers, there is no global clock, only causal order on events
Events include

-local action
-send a message
-receive a message

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proofs

Events with Order (EOrder)

Here is the signature of events with order. We need the large type Dis of discrete small types (those with decidable equality). Let Dis be this large type.

E: Dis

Loc: Dis

pred?: E → E → E + Loc

sender?: E → E + Unit

EOrder Axioms

Axiom 1: For any event e that emits a signal, we can find an event e' by which e is received.

Axiom 2: The predecessor function, pred?, is injective (one-to-one).

Axiom 3: The predecessor relation, x L y, is strongly well founded, where x L y iff for y not the first event x = pred?(y) or x = sender?(y). Namely, there is a function from E to Nat such that x L y implies $f(x) < f(y)$.

Progression of Event Structures

We progressively define the following richer structures

EOrder – events with causal order

EValue – events have values, val(e)

EState – locations have state, temporal operators

initially, when, after

ECom – communication topology is given by a graph

Etime – real time is added

ETrans – transition function (for security applications).

Event Structures (with state)

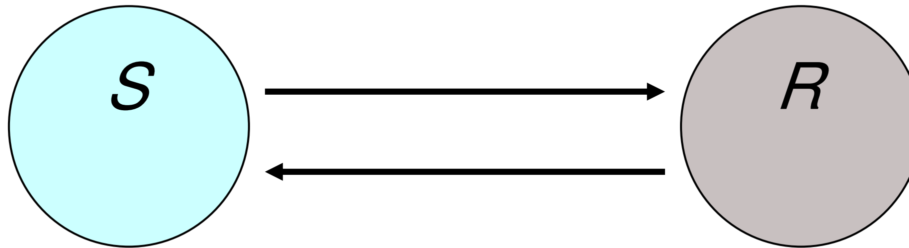
Discrete types: Events (E), Loc, Actions (Act), Tag, Link (L)

```
loc: E → Loc
kind: E → KND
first: E → B
pred: e:{x:E|¬first(x)} → {x:E|loc(x)=loc(e)}
snder: {x:E|kind(x)=rcv} → E
<:E → E → B
Ty: E → Type
val: x:E → Ty(x)
T: x:Id → i:Loc → Type
```
when : x : Id → e : E → T(loc(e), x)

after : x : Id → e : E → T(loc(e), x)

initial : x : Id → i : Loc → T(i, x)

Example – Two-Phase Handshake



$$E_p = \{e : E \mid loc(e) = p\}$$

$$Snd_{p,l} = \{e : E_p \mid sends(e, l) \neq nul\}$$

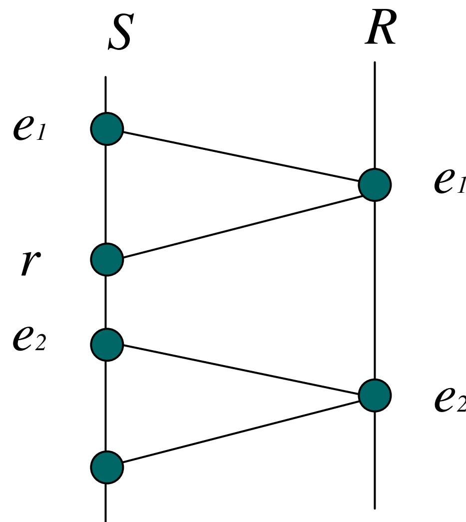
$$Rcv_{p,l} = \{e : E_p \mid kind(e) \text{ is receive on } l\}$$

Deriving the Two-Phase Handshake

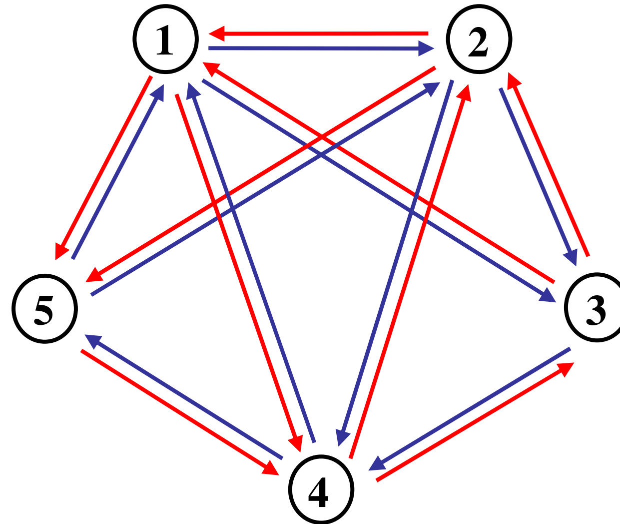
We illustrate this process by deriving a protocol for the two-phase handshake from a proof that its specification is realizable.

(1) $\forall e_1, e_2 : Snd_{S,l_1}. \quad \exists r : Rcv_{S,l_2}. \quad (e_1 < e_2 \Rightarrow e_1 < r < e_2)$

(2) $\forall e_1, e_2 : Snd_{R,l_2}. \quad \exists r_1, r_2 : Rcv_{R,l_1}. \quad (e_1 < e_2 \Rightarrow r_1 < e_1 < r_2 < e_2)$



Picture of a Computation



loc(1)	loc(2)	loc(3)	loc(4)	loc(5)
$\langle s_1, a_1, m_1 \rangle @ t$	$\langle s_2, a_2, m_2 \rangle @ t$	$\langle s_3, a_3, m_3 \rangle @ t$	$\langle s_4, a_4, m_4 \rangle @ t$	$\langle s_5, a_5, m_5 \rangle @ t$
$\langle s'_1, a'_1, m'_1 \rangle @ t+1$	$\langle s'_2, a'_2, m'_2 \rangle @ t+1$	$\langle s'_3, a'_3, m'_3 \rangle @ t+1$	$\langle s'_4, a'_4, m'_4 \rangle @ t+1$	$\langle s'_5, a'_5, m'_5 \rangle @ t+1$

Executions of Distributed Systems

Executions of distributed systems are event structures in a natural way.

In Mark's Feasibility Theorem, executions are:

At each moment of time, a process at i is in a state, $s(i, t)$, and the links are lists of tagged messages, $m(l, t)$. At each locus i and time t, there is an action, $a(i, t)$, taken. The action can be null, i.e., no state change, no receives, hence no sends.

The execution of a process (Message Automaton) is the set of all event structures consistent with it.

Fair-Fifo Executions

We assume executions are fair: channels are loss-less; and fifo: messages are received in the order sent.

1. Only the process at i can send messages on links originating at i.
2. A receive action at i must be on a link whose destination is i and whose message is at the head of the queue on that link.
3. There can be null actions that leave a state unchanged between t and $t+1$.
4. Every queue is examined infinitely often, and if it is nonempty, a message is delivered.
5. The precondition of every local action is examined infinitely often and if true the action is taken.

Outline

- Introduction
- Event Structures
- Process Extraction from Proofs

Recall Functional Program Synthesis

$\vdash \forall x{:}A.\ \exists y{:}B.\ R(x, y)$ ext $C(g_1, g_2)$

$H_1 \vdash G_1$, ext g_1 $H_2 \vdash G_2$ ext g_2

Refinements for Programs

$\vdash \forall x : A.\ \exists y : B.\ R(x, y)$ ext $\lambda x.\ \mathrm{cut}(x, z.g(x, z) ; l(x))$

$x : A \vdash \exists y : B.\ R(x, y)$ ext $\mathrm{cut}(x, z.g(x, z) ; l(x))$

by cut L

1. $x : A, z : L \vdash \exists y : B.\ R(x, y)$ ext $g(x, z)$

   by D $\ulcorner g(x, z) \urcorner$

   $x : A, z : L \vdash R(x, g(x, z))$

2. $x : A \vdash L$ ext $l(x)$

   by ____

Refinements for Systems

$\vdash \exists D : \mathtt{System}.\ \forall es : ES(D).$

$R(es)\ \text{ext}\ \ Comp(pf_1(D_1, es_1), pf_2(D_2, es_2))$

by Comp

1. $D_1 : System(G, Loc, Lnk)$

   $es_1 : ES(D_1) \vdash R_1(es_1)\ \ \text{ext}\ \ pf_1(D_1, es_1)$

2. $D_2 : System(G, Loc, Lnk)$

   $es_2 : ES(D_2) \vdash R_2(es_2)\ \ \text{ext}\ \ pf_2(D_2, es_2)$

Two-Phase Handshake Theorem

Theorem:

$\forall e_1, e_2 : Snd_s . e_1 < e_2 \Rightarrow \exists r : Rcv_s . e_1 < r < e_2$

What are the consequences of $e_1 < e_2$?

S sent two messages

Can we infer further consequences?

Relate send events to knowledge, create a boolean variable *rdy*

- Require *rdy* to be true when a send event occurs
- Require *rdy* to be false after a send event e

Two-Phase Handshake

Theorem:

$\forall e_1, e_2 : Snd_s . e_1 < e_2 \Rightarrow \exists r : Rcv_s . e_1 < r < e_2$

How to establish this by reasoning?

Suppose $e_1 < e_2$ are sends on link ℓ from S,

Then by L_1 (rdy after e_1) = false.

Since e_2 is a send, rdy must be true at e_2 by L_2.

Therefore some event e' before e_2 and after e_1

must set rdy to true.

By L_3 the event e' must be received from R on link ℓ'.

Let r be this e'.

If we have the lemmas, then the theorem is true.

Lemmas

L1. If $\texttt{S}$ sends on link ℓ then it waits

$\forall e{:}\texttt{Snd}_{S,\ \ell}\ (\texttt{rdy}\ \text{after}\ e) = \text{false}$

L2. $\texttt{S}$ sends on link ℓ only when $\texttt{rdy} = \text{true}$

$\forall e{:}\ \texttt{Snd}_{S,\ \ell}\ (\texttt{rdy}\ \text{when}\ e) = \text{true}$

L3. After $\texttt{S}$ on link ℓ sends it is ready only after an acknowledgement on link ℓ'

$\forall e : \texttt{Snd}_{S,\ \ell}\ e$ changes $\texttt{rdy}$ to true $\Rightarrow$

e is a receive from $\texttt{R}$ on link ℓ'.

Realizing the Lemmas - 1

L1. $\forall e : \mathtt{Snd}_{s,\ell} . (\mathtt{rdy} \text{ after } \mathtt{e}) = \text{false}$

We can require that the sends on ℓ are caused by the action of setting rdy to false.

a: rdy:= false; send (ℓ, <tag, m >)

We also require that only action a can send on ℓ.

This is a frame condition.

L2. $\forall e{:}\mathtt{Snd}_{s,\ell} . (\mathtt{rdy} \text{ when } \mathtt{e}) = \text{true}$

We require the precondition on the action a that rdy = true

a: pre(rdy = true) $\Rightarrow$ rdy:= false; send (ℓ, < tag, m >)

Realizing the Lemmas - 2

L3: $\forall e : \mathtt{Snd}_{s,\ell} . (e \,\Delta\, \mathtt{rdy}) = \mathtt{true}) \Rightarrow$

$\mathrm{kind}(e) = \mathtt{rcv}$ on ℓ'

We stipulate that a receive sets rdy to true

and that only a rcv or a can change rdy.

b: rcv(ℓ', m) $\Rightarrow$ rdy:= false

only [a,b] affect rdy (frame condition)

Handshake Message Automation

action local (a) sends on ℓ_1 < $\texttt{tag}$, $\texttt{v}$ >

only [a] sends on ℓ_1

state $\texttt{rdy}$: $\mathbb{B}$; initially rdy = true

precondition a is rdy = true

effect local (a) rdy := false

action $\texttt{rcv}_{\ell_2}$ < $\texttt{ack}$ > : $\texttt{Atom}$

effect rdy := true

only [local (a), $\texttt{rcv}_{\ell_2}$ <ack>] affect rdy

Question

Is the theorem true,
if we extend the automaton?

Outline

1. Introduction
2. Event Structures
3. Process Extraction from Proof

   Examples – two phase handshake

   -- leader election in a ring

   Realizers

   Realizability Theorems

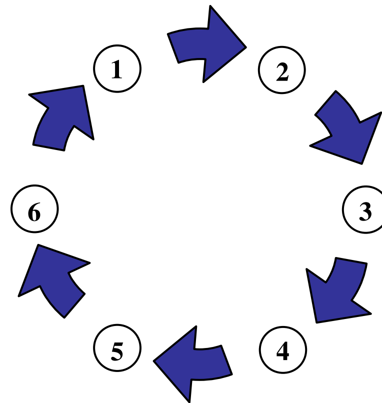
Specification for Leader Election in a Ring

Leader Election

In a Ring R of Processes with Unique Identifiers (uid's)

Specification

Let R be a non-empty list of locations linked in a ring



Let $n(i) = dst(out(i))$, the next location

Let $p(i) = n^{-1}(i)$, the predecessor location

Let $d(i,j) - \mu k \geq 1.\ n^{k}(i) = j$, the distance from i to j

Note $i \neq p(j) \Rightarrow d(i,p(j)) = d(i,j)-1$.

Specification, continued

$$\text{Leader}(R,es) == \exists\, ldr{:}\, R.\ \big(\exists e@ldr.\ kind(e)=leader\big)\ \&\ \big(\forall i{:}R.\ \forall e@i.\ kind(e)=leader \Rightarrow i=ldr\big)$$

Theorem

$\forall$ R:List(Loc). Ring(R)

$\exists$ D:Dsys(R). Feasible(D) &

$\forall$es: ES. Consistent(D,es). Leader(R,es)

Logically Decomposing the Leader Election Task

Let LE(R,es) == $\forall$i:R.

1. $\exists$e. kind(e)=rcv(out(i), <vote,uid(i)>)

2. $\forall$e. kind(e)=rcv (in(i), <vote,u>) $\Rightarrow$

   (u>uid(i) $\Rightarrow$ $\exists$e′.kind(e′)=rcv (out(i),<vote,u>))

3. $\forall$e′. [(kind(e′)=rcv(out(i), <vote,uid(i)>)) $\vee$

   $\exists$e. (kind(e)=rcv(in(i), <vote,u>)& (e < e′ & u> uid(i)))]

4. $\forall$e@i. kind(e)=rcv(in(i),uid(i)). $\exists$e′@i. kind(e′)=leader

5. $\forall$e@i. kind(e)=leader. $\exists$e@i. kind(e)=rcv (in(i), <vote,uid(i)>)

Realizing Leader Election

Theorem $\forall \texttt{R:List(Loc)}\ .\ \texttt{Ring(R)}$

$\exists \texttt{D:Dsys(R)}\ .\ \texttt{Feasible(D)}\ .$

$\forall \texttt{es:Consistent(D, es)}\ .\ \left(\texttt{LE(R, es)} \Rightarrow \texttt{Leader(R, es)}\right)$

Proof: Let $\texttt{m} = \texttt{max}\ \{\texttt{uid(i)} \mid \texttt{i} \in \texttt{R}\}$, then $\texttt{ldr} = \texttt{uid}^{-1}\texttt{(m)}$.

We prove that $\texttt{ldr} = \texttt{uid}^{-1}\texttt{(m)}$ using three simple lemmas.

Intuitive argument that a leader is elected

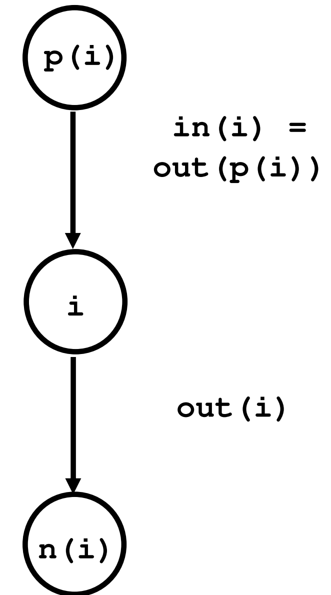
1. Every i will get a vote from predecessor for the predecessor.

2. When a process i gets a vote u from its predecessor with u with $u >$ uid(i) it sends it on.

3. Every rcv is either vote of predecessor rcvin(i) for itself or itself or a vote larger than process id before.

4. If a processor sets a vote for itself, it declares itself ldr. ldr.

5. If a processor declares ldr it got a vote for itself.



Lemmas

Lemma 1. $\forall \texttt{i} : \texttt{R}.\ \exists \texttt{e} @ \texttt{i}.\ \texttt{kind(e)} = \texttt{rcv}\ (\texttt{in(i)}, \texttt{<vote, ldr>})$

By induction on distance of i to ldr.

Lemma 2. $\forall \texttt{i}, \texttt{j} : \texttt{R}.\ \forall \texttt{e} @ \texttt{i}.\ \texttt{kind(e)} = \texttt{rcv}\ (\texttt{in(i)}, \texttt{<vote, j>})\ .$

$$(\texttt{j} = \texttt{ldr} \lor \texttt{d(ldr, j)} < \texttt{d(ldr, i)})$$

By induction on causal order of rcv events.

Lemma 3. $\forall \texttt{i} : \texttt{R}.\ \forall \texttt{e}' @ \texttt{i}.\ (\texttt{kind(e}'\texttt{)} = \texttt{leader} \Rightarrow \texttt{i} = \texttt{ldr})$

If $\texttt{kind(e}'\texttt{)} = \texttt{leader}$, then by property 5, $\exists \texttt{v} @ \texttt{i}.\texttt{rcv}\ (\texttt{in(i)}, \texttt{<vote, uid(i)>})\ .$

Hence, by Lemma 2 $\texttt{i} = \texttt{ldr} \lor (\texttt{d(ldr, i)} < \texttt{d(ldr, i)})$

but the right disjunct is impossible.

Finally, from property 4, it is enough to know

$\exists \texttt{e}.\texttt{kind(e)} = \texttt{rcv}\ (\texttt{in(ldr)}, \texttt{<vote, uid(ldr)>})$

which follows from Lemma 1.

QED

Realizing the clauses of LE(R,es)

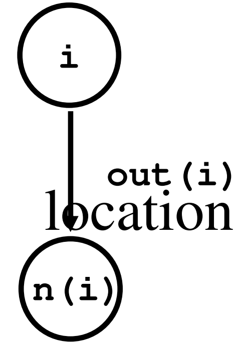
We need to show that each clause of `LE(R,es)` can be implemented by a piece of a distributed system, and then show the pieces are compatible and feasible.

We can accomplish this very logically using these Lemmas:

- Constant Lemma
- Send Once Lemma
- Recognizer Lemma
- Trigger Lemma

Implementing Clause 1 of LE(R,es)

1. ∃e.kind(e) = rcv (out(i) , <vote, uid(i)>)



We need to send <vote,uid(i)> from each location i.

The Constant Lemma allows us to create a state variable me at each i with me=uid(i)

∀e @ i. (me **when** e) = uid(i)

The Send Once Lemma lets the process at i send uid(i)

∃e.kind(e) = rcv (out(i) , <vote, uid(i)>)

Implementing Clauses 4, 5 of LE(R,es)

We can instantiate the Trigger Lemma to obtain

$$
\begin{array}{l}
\forall \texttt{i : Loc.} \\
\quad \forall \texttt{e}' \,@\, \texttt{i.kind(e}') = \texttt{leader.} \\
\qquad (\exists \texttt{e} \,@\, \texttt{i. e} < \texttt{e}'\texttt{. kind(e) = rcv(in(i), <vote, uid(i)>)}) \\
\qquad \forall \texttt{e} \,@\, \texttt{i. kind(e) = rcv(in(i), <vote, uid(i)>)} \Rightarrow \\
\qquad\quad \exists \texttt{e}' \,@\, \texttt{i. kind(e}') = \texttt{leader}
\end{array}
$$

Leader Election Message Automaton

state $me : \mathbb{N}$; initially $uid(i)$

state $done : B$; initially $false$

state $x : B$; initially $false$

action $vote$; precondition $\neg done$

effect $done := true$

sends $[msg(out(i), vote, me)]$

action $rcv_{in(i)}(vote)(v) : \mathbb{N}$;

sends if $v > me$ then $[msg(out(i), vote, v)]$ else[]

effect $x :=$ if $me = v$ then $true$ else x

action $leader$; precondition $x = true$

only $rcv_{in(i)}(vote)$ affects x

only $vote$ affects $done$

only $\{vote, rcv_{in(i)}(vote)\}$ sends $out\ (i), vote$

Outline

- Introduction
- Event Structures
- Process Extraction from Proof

  Examples – two phase handshake

  -- leader election in a ring

  Realizers

  Realizability Theorems

Realizable specifications

initial -- gives the value of a variable

effect -- defines a change of state based on an action

frame -- limits actions that can change a variable

pre -- takes an action if a precondition is true

sends -- sends tagged messages on a specified link

sframe -- limits actions that can send

Realizing Primitive Event Specifications

initial (using Message Automata)

@i state x:T; initially p(x)

realizes : p(initial (x,i))

frame

Define $x_\vartriangle e$ as x after e $\neq$ x when e

@i only L affects x

realizes : $\forall e@i. (x_\vartriangle e \Rightarrow kind(e) \in L)$

Effect lemma

effect

@i state $x:T_1$; action $k:T_2$;

k(v) effect x: = f(s when e, val (e))

realizes

$\forall$e@i. kind(e)=k $\Rightarrow$

x after e = f(s when e, val (e))

pre

@i action k:T; k(v) precondition p(s,v)

realizes:

$$\forall e@i.\ (\mathtt{kind}(e) = k \Rightarrow p(s\ \mathtt{where},\ \mathtt{val}(e)))$$

$$\&\ \forall e@i.\ \exists e'@i.\ e \leq e'\ \&$$

$$(\mathtt{kind}(e') = k \lor \forall v : T.\ \neg p(s\ \mathtt{after}\ e',\ v))$$

$$\&\ \exists v : T.\ p(\mathtt{init}(es)(i),\ v) \Rightarrow \exists e : E.\ \mathtt{loc}(e) = i$$

The skolem function in the $\forall e\ \exists e'$ clause gives a "schedule" for the action k.

Compound Realizers

Realizers are built by combining the six basic clauses. In the concrete case of Message Automata, the clauses are just joined by union. In the abstract setting, there is a combining operator, $R_1 \oplus R_2$.

es-realizer

Realizer

$\equiv_{\text{def}}$ rec(X.Unit

$+X \times X$

$+\text{Id}\times T\text{:Type}\times\text{Id}\times T$

$+\text{Id}\times\text{Type}\times\text{ID}\times(\text{Knd List})$

$+\text{IdLnk}\times\text{Id}\times(\text{Knd List})$

$+\text{Id}\times ds{:}x\text{:Id fp}\to\text{Type}\times\text{Knd}\times T\text{:Type}\times x\text{:Id}\times\left(\text{State}(ds)\to T\to\text{DeclaredType}(ds;x)\right)$

$+ds{:}x\text{:Id fp}\to\text{Type}$

$\times\text{Knd}$

$\times T\text{:Type}$

$\times\text{IdLnk}$

$\times dt{:}x\text{:Id fp}\to\text{Type}$

$\times\left((tg\text{:Id}\times(\text{State}(ds)\to T\to(\text{DeclaredType}(dt;tg)\text{ List})))\text{ List}\right)$

$+\text{Id}\times ds{:}x\text{:Id fp}\to\text{Type}\times\text{Id}\times T\text{:Type}\times(\text{State}(ds)\to T\to\text{Prop})$

Compatibility

Arbitrary compositions, $R_1 \oplus R_2$ might not be compatible. For example, R_1 might be a frame condition that says

R_1: only action k can change x and

R_2: action k' changes x for some $k \neq k'$

Also compatible realizers must have compatible types

R_1: declares x to be of type T_1

R_2: declares x to be of type T_2

We must have $T_1 \sqsubseteq T_2$ or $T_2 \sqsubseteq T_1$

Compatibility, continued

Compatibility is defined by 15 conditions from the 6 by 6 matrix of possibilities (half minus the diagonal). They are not decidable in theory but are in practice.

Outline

- Introduction
- Event Structures
- Process Extraction from Proof

    Examples – two phase handshake

    -- leader election in a ring

    Realizers

    Realizability Theorems

Consistency

If `P` is any event specification, then the type theory expression of the goal is this

$$\vdash \exists D : \texttt{DSyst}.\ \texttt{Feasible}(D)\ \&\ \forall es{:}ES.\ \texttt{Consistent}(D, es) \Rightarrow P(es)$$

We say that `Feasible(D)` if `D` has at least one execution.

We say `Consistent(D,es)` provided `es` is an event system that arises from a possible execution of `D`.

Feasibility

A realizer R is feasible if it has an execution. For this to be possible, the clauses of R must be compatible and the types of variables, event values, and message content must be nonempty.

Computability

One of the main theorems of Bickford's massive library is that if distributed system D is feasible, then we can construct the possible executions, worlds, of it.

Moreover, from a world `W of D`, we can construct event systems for `D`, `es(D)`, consistent with it.

```
Consistent(D,es)
```

This is a constructive proof, as are all in the library. So it defines the computational rules for the realizers given a schedule.

Running Distributed Systems Generate Event Structures

Theorem 1

For all DSys D, Feasible(D) $\Rightarrow \exists$w:World. Possible(D,w)

Theorem 2

For all DSys D and all Possible Worlds w of D,

we can build an EventStructure es(w) Consistent$\big($D, es(w)$\big)$.

Logic of Events, circa 2007

What distinguishes our event structures approach?

-based directly on Leslie Lamport's insights;

type theory captures them naturally

-used by distributed computing researchers, matches their their intuitions

-integrated into LPE, hence into procedural programming

-completely formalized

-supports proofs-as-processes of synthesis and programming programming

-widely applicable: verification, optimization, documentation, documentation, security, performance

-organizes a fundamental set of concepts

Exercise

Specify that a group of processes all have the same function for integer root in their state.

Message Automata Clauses

- @ i x : T **initially** = v
- @ i **effect** k(v : t) **on** x

  x := f state f
- @ i **precondition** a(v:t) **is** P state v
- k(v : T) **sends on link l**

  [tg_1, f_1 state v; . . .; tg_n, f_n state v]
- @ i **only** [k_1, . . . , k_n] **sends on link l with tag** tg
- A ⊕ B, **where** A, B **are message automata**

Citation

On urelements in computation

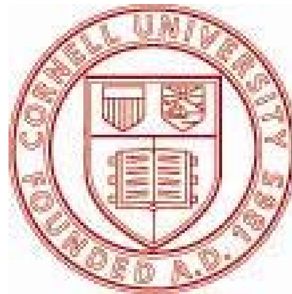
Andreas Blass, Yuri Gurevich, and Saharori Shelah, Choiceless Polynomial Time, *Annals of Pure and Applied Logic*, 100, 1999, 141-187.

Formal Basis for Security Properties

Lecture 4

Robert L. Constable

Joint with Mark Bickford



Marktoberdorf, Germany – Summer 2007

Reliability, Correctness, and Security

Reliability and correctness are properties of software that make sense in even the simplest computer models – functional code on one processor.

Security is a sensible property for the distributed computing model of which the Internet is an instance.

Secure Communication

A building block concept for security properties on Internet computing systems is secure communications, i.e. process **A** can send content in messages to process **B** that no other process learns.

Typically this property is achieved by encrypting the messages in such a way that only **B** can encrypt them, thus a process **C** that intercepts the message does not learn the content.

Essential Elements of Security Models

Agents

Protected information content

Learning content

Outline

Motivation

- security models
- a new model

Unguessable Atoms

- Type Atom (Urelements)
- Properties
- Permutation Rule

Independence

- Content does not involve Atom a
- Rules
- Applications (nonces)

Conclusion to Series

Bickford's Analysis

There are two kinds of security models

Analytic	Algebraic
learning $\neq$ Acquiring	to learn is to acquire
general computation	restricted computation

Analytic Model

Computation system includes all computer programs.

Agents can guess content, so learning is not the same as acquiring. Learning definitions can depend on resource bounds of agents, thus on computational complexity, and on probability assumptions about "cracking a code."

Algebraic Model (Dolev-Yao ’83)

To learn the content is to acquire it or generate it – guessing is disallowed.

Protected information is a finitely generated algebra over atomic pieces (keys, nonces, atomic messages).

New Security Model

Mark Bickford proposed a way to use elements of Type Theory inspired by our digital library work to create a new security model which combines a generally model of computation with a simple model of learning protected information, namely

all programs

learning is acquiring.

How is this possible?

In a general model of computation, agents can guess secrets by enumerating all possible content.

How to prevent that!

Answer: protected content is built from unguessable atoms.

What is an unguessable atom?

Here are the properties we want.
They are elements of a primitive type, in fact the CTT type Atom, included since 1984 but with an additional rule. (A major advantage of the open-ended nature of CTT.) The only operations on atoms is to

compare them for identity

They cannot be generated or constructed from other elements. Otherwise, they behave as ordinary data elements.

Reading about Atoms in Nuprl

In the working material, the paper by Stuart Allen entitled: ***An Abstract Semantics for Atoms in Nuprl***, provides the semantics for Atoms and explains why they are "unguessable."

It is only 10 pages of content.

I hope you will read pages 4 – 7.

Atoms and Urelements

The closest concept in Set Theory to Atoms is the notion of urelement. These are atomic non-set primitive elements. Some accounts of them might be useful in a classical account of security.

Sets	Types
Urelements	Atoms

Citation

On urelements in computation
Andreas Blass, Yuri Gurevich, and Saharori Shelah, Choiceless Polynomial Time, *Annals of Puter and Applied Logic*, 100, 1999, 141-187.

Properties of Atoms

- Atoms are not enumerable, not infinite. Yet they are not of a fixed finite cardinality.
- The canonical elements are ***token{i:ut}*** where the class assigned to an unhideable token is a parameter ***D***, some discrete class.
- The semantics quantifies over all possible values for a discrete ***D*** (this is a supervaluation semantics).
- ***token{i:ut}*** reduces to itself, i.e. is canonical.
- Equality is decidable on Atom
- For any ***k***, we can pick out ***k*** atoms from Atom.

Properties of Atoms continued

- For any character string a, ***token{a:ut}*** is a possible instance of the semantics (see Allen p.6, $True_{+}$).
- If the elements of **D** are unhideable in definitions (occur on both left and right sides) and if the evaluation rules respect permutation of the names in **D**, then any sequent **J** true of **k** atoms, **J (a,b,c,…),** is true for any permutation **a** to $\mathbf{a^1}$, **b** to $\mathbf{b^1}$, …, i.e. $\mathbf{J(a^1,b^1,c^1,\ldots)}$.

Examples

$f(x) = =$ **if** $x=1$ then *token(a)* **else** *token(b)* **fi**

is not a legal definition because a, b do not appear on the left hand side.

$f\{a,b\}(x) = =$ …would be legal.

It is illegal to introduce a term, say *oups,* and have a rule *oups* ↓ *token(a)*.

Outline

Motivation

- security models
- a new model

Unguessable Atoms

- Type Atom (Urelements)
- Properties
- Permutation Rule

Independence

- Content does not involve Atom a
- Rules
- Applications (nonces)

Conclusion to Series

Tracking atoms in a distributed computation

We want to track when a process acquires an unguessable atom. It does so when its state depends on that atom. This happens only because the state initially has the atom or because it was received in a message since it is not possible for a process to construct an atom it does not already have.

We express this idea by talking about a state being independent of an atom up to some receive event.

Independence Rules

Here are the key rules for the proposition $(x{:}t \parallel a)$ which expresses "*x* of type *T* is independent of atom a."

Triviality (*t* has no atoms)

$H \vdash (t{:}T \parallel a)$

$H \vdash t \in T$

$H \vdash a \in \text{Atom}$

t is closed and mentions

no tokens

Independence Rules

Base (*t* is a different atom)

$$H \vdash (t: Atom \| a)$$

$$H \vdash \neg(t = a \text{ in } Atom)$$

Application (key rule)

$$H \vdash (f(t):B[t/v] \; \| a\,)$$

$$H \vdash (\,f:(v:A \rightarrow B \; \| a\,)$$

$$H \vdash (t:A \| a)$$

Independence Rules

Absurdity (*a* depends on *a*)

$$H \vdash \neg\, (a{:}Atom \parallel a)$$

Set (separating predicate is irrelevant)

$$H \vdash (t{:}\{x{:}T|P\} \parallel a)$$

$$H \vdash (t{:}T \parallel a)$$

$$H \vdash t \ in \ \{x{:}T \mid P\}$$

Independence Rules

Equality

$$H \vdash (t_1{:}T_1 \| a_1) = (t_2{:}T_2 \| a_2)$$

$$H \vdash T_1 = T_2 \textit{ in Type}$$

$$H \vdash t_1 = t_2 \textit{ in } T$$

$$H \vdash a_1 = a_2 \textit{ in Atom}$$

This allows us to build up independence from pieces.

Basic Facts

We can prove general facts such as
Theorem $\forall \boldsymbol{a} : \boldsymbol{Atom}. \forall z : \mathbb{Z}.(z : \mathbb{Z} \| \boldsymbol{a})$

We proceed by induction on I (upwards and downwards) starting from the fact that 0 is a closed term with no atoms. Going upwards, if $(z : \mathbb{Z} \| \boldsymbol{a})$, then $(z+1 : \mathbb{Z} \| \boldsymbol{a})$ since $z+1$ is

$\lambda(\boldsymbol{x}.\boldsymbol{x}+1)(z)$ and $\lambda(\boldsymbol{x}.\boldsymbol{x}+1)$
is a closed term with no atoms
and the application rule applies
Qed.

Outline

Motivation

- security models
- a new model

Unguessable Atoms

- Type Atom (Urelements)
- Properties
- Permutation Rule

Independence

- Content does not involve Atom a
- Rules
- Applications (nonces)

Conclusion to Series

Specifying Security Properties

Here is a way to specify security properties depending on a cryptographic service. First we specify the service with a predicate P_{crypto} and the security property with Q_{secure}. Then we show that some message automaton M realizes $P_{crypto} \Rightarrow Q_{secure}$.

$$\vdash (P_{crypto} \Rightarrow Q_{secure}) \text{ extract } M$$

A Security Property

Suppose we want to say that a group of agents will share a common secret. Let the agents be at locations on the list $\boldsymbol{L}$.

The safety specification is that the state of any agent not on $\boldsymbol{L}$ is always independent of a secret $\boldsymbol{a}$.

$$\boldsymbol{Q}_1(\boldsymbol{a}) == \forall \boldsymbol{e}.[\boldsymbol{loc}(\boldsymbol{e}) \in \boldsymbol{L} \vee ((\boldsymbol{state}(\boldsymbol{loc}(\boldsymbol{e}))\boldsymbol{when}\ \mathrm{e}) \| \boldsymbol{a})]$$

The liveness part is that eventually all agents on $\boldsymbol{L}$ receive the secret.

$$\boldsymbol{Q}_2(\boldsymbol{a}) == \forall \boldsymbol{i}.\ \boldsymbol{i} \in \boldsymbol{L} \Rightarrow \exists \boldsymbol{e}.(\boldsymbol{loc}(\boldsymbol{e}) = \boldsymbol{i}\ \&\ (\boldsymbol{x}\ \text{when}\ \boldsymbol{e}) = \boldsymbol{a})$$

The full specification is

$$\exists \boldsymbol{a} : \boldsymbol{Atom}.\boldsymbol{Q}_1(\boldsymbol{a})\ \&\ \boldsymbol{Q}_2(\boldsymbol{a})$$

Cryptographic Services

Mark Bickford and **Robbert van Rennesse** have modeled a Public Key Cryptography service in the Logic of Events using atoms. It is sketched in a forthcoming article by Bickford:

> ***Unguessable Atoms***: A Logical Foundation for Security Supported by the Information Assurance Institute at Cornell.

We will look at a simpler service, providing nonces.

A simple nonce service

We can equip Message Automata to choose nonces by taking a nonce to be a distinct unguessable value, an atom. One way to do this is to assume that every agent has a supply of atoms unique to itself. We call this the

Nonce Assumption

$$\forall \boldsymbol{i}, \boldsymbol{j} : \boldsymbol{Loc}.\ \boldsymbol{i} \neq \boldsymbol{j} \Rightarrow \forall \boldsymbol{a} : \boldsymbol{Atom}.\boldsymbol{a} \in (\text{Nonces initially } \boldsymbol{i}) \Rightarrow (\boldsymbol{j} \| \boldsymbol{a})$$

Implementing Atoms

To generate real code from the Message Automata that use Atoms to provide cryptographic services, we would resort to the standard technique of using random bit strings or RSA style public key that depend on computational complexity results and one-way functions.

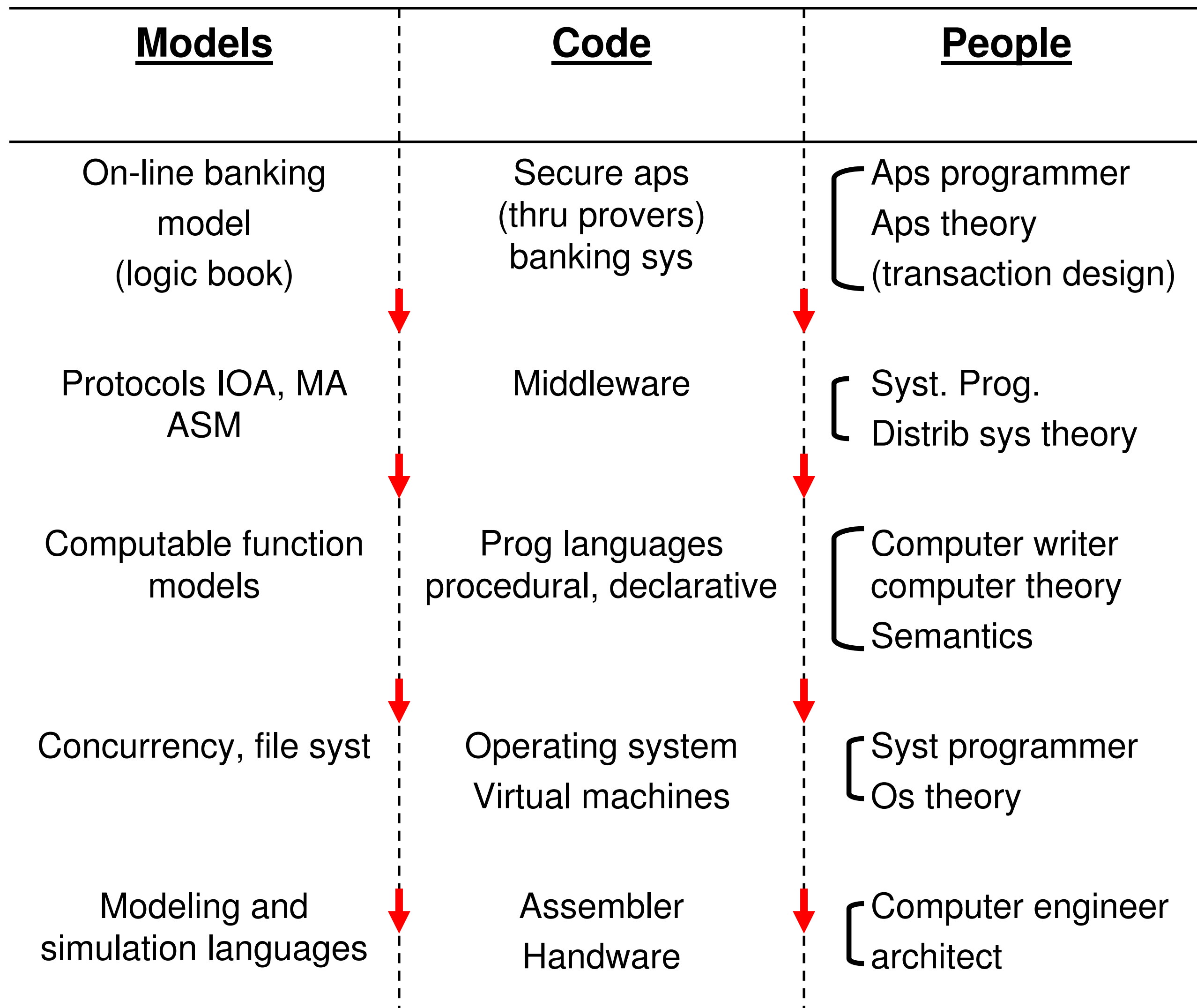
Series Conclusion

We have shown how public trust in information technology relies on a partnership between computer scientists and mathematicians - especially logicians.

The computer scientist must abstract away detail to present concepts that are mathematically tractable yet faithful to computing practice and capable of information and guiding the technology. For example:

- The digital abstraction
- Automata and state machines
- Induction and recursion
- Computability
- Data and types
- Formal correctness
- Asynchronous distributed computing
- Event structures

Computer Science Practice – "the stack", translators



This is how we span levels of abstraction

Series Conclusion Continued

We have presented a correct-by-construction refinement methodology for distributed computing, a long standing challenge, and a capability more critical than in the functional case.

We have integrated a logic for the event structure abstraction into the comprehensive theory of computing serving computer science, and have shown how to include security properties in a novel and formally tractable way.