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
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

Series Introduction and Type Theory Basics
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

<table>
<thead>
<tr>
<th>Models</th>
<th>Code</th>
<th>People</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-line banking model &lt;br&gt; (logic book)</td>
<td>Secure aps (thru provers) banking sys</td>
<td>Aps programmer &lt;br&gt; Aps theory &lt;br&gt; (transaction design)</td>
</tr>
<tr>
<td>Protocols IOA, MA ASM</td>
<td>Middleware</td>
<td>Syst. Prog. &lt;br&gt; Distrib sys theory</td>
</tr>
<tr>
<td>Computable function models</td>
<td>Prog languages procedural, declarative</td>
<td>Computer writer &lt;br&gt; computer theory &lt;br&gt; Semantics</td>
</tr>
<tr>
<td>Concurrency, file syst</td>
<td>Operating system &lt;br&gt; Virtual machines</td>
<td>Syst programmer &lt;br&gt; Os theory</td>
</tr>
<tr>
<td>Modeling and simulation languages</td>
<td>Assembler &lt;br&gt; Handware</td>
<td>Computer engineer architect</td>
</tr>
</tbody>
</table>

This is how we span levels of abstraction
How the stacks are grounded

<table>
<thead>
<tr>
<th>Models</th>
<th>Code</th>
<th>People</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS Theory Mathematics (Set Theory)</td>
<td>Hardware Physics</td>
<td>Computer Science &amp; Engineering</td>
</tr>
</tbody>
</table>
Refining the Questionnaire

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. \( \lambda \) calculus \( \rightarrow \) Lisp

  Simple Theory of Types \( \rightarrow \) 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 carry out 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 into 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.
For formalization, **detail matters** – too much detail for most people. Consider the simple examples above.

**Factoring**

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

or

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

or

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

where \( \text{prod} (\text{nil}) = 1 \).
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, $\text{Inf}$, i.e.

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

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

$$\phi \in \text{Inf} \quad \frac{x \in \text{Inf}}{s(x) \in \text{Inf}}$$

Such a set is inductive.
Computing with Numbers

Rules for computing are in a metalanguage

\[ x \cdot 0 = 0 \]
\[ x \cdot (n+1) = (x \cdot n) + x \]

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

\[ 0 \in \mathbb{N} \text{ and if } n \in \mathbb{N} \text{ then } s(n) \in \mathbb{N} \]

or the elements are

\[
\text{nat}\{n : \text{number}\} \quad \text{n a decimal numeral}
\]

the noncanonical numbers are expressions that reduce to canonical numbers,

\[ \text{e.s. } 2 + 2, 3!, 2^3, \text{ etc.} \]

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 $\forall n : \mathbb{N}. \exists F : (Prime)List. \text{prod}(F) = n$
tells us that there is a function

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

$$\text{factor}(1) = \text{nil}$$
$$\text{factor}(2) = \{2\}$$
$$\text{factor}(3) = \{3\}$$
$$\text{factor}(4) = \{2, 2\}$$

What is a function from $A$ to $B$?

informal mathematics: a rule of correspondence from $A$ to $B$
Set Theory:

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

$$F \subseteq P(A \times B)$$

such that

$$\forall x \in A. \exists y : B. <x, y> \in F \ & \ \forall p, q : F \text{ if } \text{fst}(p) = \text{fst}(q) \text{ then } \text{snd}(p) = \text{snd}(q)$$

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.
Here is multiplication in type theory,

A lambda term using a fixed point combinator

\[ \text{fix}(f \cdot \lambda(x, y. \; \text{if } x = 0 \text{ then } 0 \text{ else } y + f(x-1, y))) \]
Questions

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

Is $\lambda(x.\lambda(y.0)) = \lambda(y.0)$?
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 $n + \text{exp}$ might diverge if $\text{exp}$ 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 polymorphic 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.

\[
\text{List}(A) = A \cup (A \times \text{List}(A))
\]

or

\[
\text{List}(A) = 1 \oplus (A \times \text{List}(A))
\]

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

\[
A \cup A \times A \cup A \times (A \times A) \cup ...
\]
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.

\[ \text{List}(A) = \mu Y . (1 \oplus A \times Y) \].

It suffices for most purposes to use one inductive type, a W-type (Martin-Löf) to encode recursive ordinals

\[ \mu W . (x : A \times (B_x \rightarrow W)) \]
Formalization

Recall the formulation
\[
\forall n : \{ i : N \mid 1 < i \} . \exists F : (\text{Prime}) \text{List}. prod(F) = n
\]
The type \( \{ i : \mathbb{N} \mid 1 < i \} \) seems like a normal set formed by separation, but in type theory it also has computational significance, namely the function

**factor: \( \{ i : \mathbb{N} \mid 1 < i \} \rightarrow (\text{Prime}) \text{List} \)**

does not take evidence for \( 1 < i \) as input.

Otherwise, the separation type \( \{ x : A \mid B \} \) is like the corresponding subset of \( A \).
Equality

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

\[ 0 = \emptyset \quad 1 = \{ \emptyset \} \quad 2 = \{ \emptyset, \{ \emptyset \} \} \quad 3 = \{ \emptyset, \{ \emptyset \}, \{ \emptyset, \{ \emptyset \} \} \} \ldots \]

This is also 0, \{0\}, \{0,1\}, \{0,1,2\}…

Notice \( \{ \emptyset, \{ \emptyset \} \} = \{ \{ \emptyset \}, \emptyset \} \) 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. $A \cap \text{Top} = A$, and $A \sqsubseteq \text{Top}$ 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 $\text{fix}(x.x)$. 
Intersection

In Set Theory  \( A \cap B = \{x : A \mid x \in B\} \)

In Type Theory:

\( a = b \) in \( A \cap B \) if \( a = b \) in \( A \) and \( a = b \) in \( B \)

\( \{x : A \mid P\} \cap \{x : A \mid Q\} = \{x : A \mid P \& Q\} \)

Notice that \( \mathbb{Z}_2 \cap \mathbb{Z}_3 = \mathbb{Z}_6 \), e.g. \( 0 = \overline{2}_6 \) and \( 0 = \overline{3}_6 \)

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

Dependent Intersection

In Type Theory \( x : A \cap B \) is the set of all elements \( x \) of \( A \) such that \( x \in B \)
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:

- $A \subseteq \text{Top}$
- $\text{Void} \subseteq A$
- $\{x:A \mid P(x)\} \subseteq A$
- $A \subseteq A//R$
- $\{1,2,3\} \subseteq \mathbb{Z} \subseteq \mathbb{Z}_6 \subseteq \mathbb{Z}_2$
Subtyping relations

If $A \sqsubseteq A'$ and $B \sqsubseteq B'$ then:

\[
\begin{align*}
A + B & \subseteq A' + B' \\
A \times B & \subseteq A' \times B' \\
A' \to B & \subseteq A \to B' \\
\{x : A \mid P(x)\} & \subseteq A \\
A \cap B & \subseteq A \\
A \cap B & \subseteq B \\
A & \subseteq A\mathbin{\mathord{/}}\mathbin{\mathord{/}}E \\
A & \subseteq A \cup B \\
A & \subseteq \text{Top}
\end{align*}
\]
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 \( A \times (B \times C) \) and the field selectors can be defined as functions on tuples, say

\[
\begin{align*}
x &= \lambda (r.1of(r)) \\
y &= \lambda (r.1of(2of(r))) \\
z &= \lambda (r.2of(2of(r)))
\end{align*}
\]
Naïve Record Extension

We can provide for record extension by adding $\text{Top}$ as a last component of any record

$$\text{Records} == T : u_i \times \text{Top}$$

We build up the previous record as follows:

$$R_1 == A \times \text{Top}$$

$$R_2 == A \times (B \times \text{Top})$$

$$R_3 == A \times (B \times (C \times \text{Top}))$$
Naïve Record Subtyping

Notice that:

\[ R_2 \subseteq R_1 \quad \text{since} \quad A \subseteq A, \quad B \times \text{Top} \subseteq \text{Top} \]

\[ R_3 \subseteq R_2 \quad \text{since} \quad A \subseteq A, \quad B \times (C \times \text{Top}) \subseteq B \times \text{Top} \]

\[ R_2 \in \text{Records} \quad \text{since} \quad A \in \cup_i, \quad (B \times \text{Top}) \in \text{Top} \]
Records Using Labels

Another approach to records is to take labels, \( L \), as indexes into components.

- **Given** \( \{ x:A; y:B; z:C \} \)
- **take** \( L=\{x,y,z\}, \ L \subseteq \text{Atom} \)
- **Define** \( \text{Sig}:L \rightarrow U_i \) by
  - if \( j=x \) then \( A \)
  - else if \( j=y \) then \( B \) else \( C \)

Define the record type as \( x:L \rightarrow \text{Sig}(x) \).
Records as Functions

We now take

\{x:A; y:B; z:C\} == x:L \rightarrow \text{Sig}(x).

For \( r \in \{x:A; y:B; z:C\} \),

let \( r.i == r(i) \)

so \( r.x \in A, r.y \in B, r.z \in C \)
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 \(L \subseteq L'\).

Define \(\text{Sig}'(i) = \text{if } i = w \text{ then } D \text{ else } \text{Sig}(i)\).

Notice \(x: L' \rightarrow \text{Sig}'(x) \subseteq x: L \rightarrow \text{Sig}(x)\)
because \(L \subseteq L'\) and \(\text{Sig}'(x) \subseteq \text{Sig}(x)\) for \(x \in L\).
Record Extension Depends on Function Polymorphism

\[ x : L' \rightarrow \text{Sig}'(x) \equiv x : L \rightarrow \text{Sig}(x) \]

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

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

<table>
<thead>
<tr>
<th>Sets</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\text{void}$</td>
</tr>
<tr>
<td>$\mathbb{N}$</td>
<td>$\mathbb{N}$</td>
</tr>
<tr>
<td>${x : A \mid P}$</td>
<td>${x : A \mid P}$</td>
</tr>
<tr>
<td>$A \times B$</td>
<td>$A \times B$</td>
</tr>
<tr>
<td>$A \cap B$</td>
<td>$A \cap B$</td>
</tr>
<tr>
<td>$A \oplus B$</td>
<td>$A + B$</td>
</tr>
<tr>
<td>$B^A$</td>
<td>$A \rightarrow B$</td>
</tr>
<tr>
<td>$\prod_{x \in A} B_x$</td>
<td>$x : A \rightarrow B_x$</td>
</tr>
<tr>
<td>$\sum_{x \in A} B_x$</td>
<td>$x : A \times B$</td>
</tr>
<tr>
<td>$\mathcal{P}(A)$</td>
<td>$\mathcal{P}_i(A)$</td>
</tr>
<tr>
<td>$\mu(x.F)$</td>
<td>$\mu(x.F)$</td>
</tr>
<tr>
<td>$A / E$</td>
<td>$A / / E$</td>
</tr>
<tr>
<td>$\Omega_k$</td>
<td>$\Omega_i$</td>
</tr>
<tr>
<td>$A \subseteq B$</td>
<td>$A \subseteq B$</td>
</tr>
</tbody>
</table>
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 \( A \times B, A \cap E \) to explain the difference between fractions and rational numbers \((\mathbb{Q})\).

• Can you define \( \mathbb{Q} \) so that \( \mathbb{Z} \subseteq \mathbb{Q} \)?

• Call a type discrete if its equality is decidable.
  • How can we say that \( \mathbb{N}, \mathbb{Q} \) are discrete?
  • Is \( \mathbb{N} \rightarrow \mathbb{N} \) discrete?

• Define: \( a @ \ell = \text{if } a = \text{nil then } \ell \)
  \[ \text{else } \text{hd}(a) \bullet (t\ell(a) @ \ell) \text{fi} \]
  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)

Set Theory
(ZFC)
Proposed Modification

Set Theory
IZF + C

or

Computation

Computational Type Theory
(evaluation rules)
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
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

\[ \forall x : A. \ R(x, f(x)) \]

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

\[ \forall x : A. \ \exists y : B. \ R(x, y) \]

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

\[ \forall x : A. R(x, f(x)) \]

\[ \forall x : A. \exists y : B. R(x, y) \]
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 $\llbracket A \rrbracket_\mathcal{M}$.

Sometimes we suppress the model.

**Truth and Evidence:**

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

$$\alpha \in \llbracket A \rrbracket_\mathcal{M} \text{ if } \models_\mathcal{M} A$$

If $A$ is false, then $\llbracket A \rrbracket = \emptyset$, and if $A$ is true, then $\llbracket A \rrbracket \neq \emptyset$. 
Propositional Evidence

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

Here is how to construct evidence for compound formulas:

\[
\begin{align*}
\left[ A \land B \right] &= \left[ A \right] \times \left[ B \right] \\
\left[ A \lor B \right] &= \left[ A \right] \oplus \left[ B \right] \\
\left[ A \rightarrow B \right] &= \left[ A \right] \rightarrow \left[ B \right] \\
\left[ \neg A \right] &= \left[ A \right] \rightarrow \phi
\end{align*}
\]
Evidence for Quantified Statements

\[ \left[ \exists x : A.B \right] x : [A] \times [B] \]

\[ \left[ \forall x : A.B \right] x : [A] \rightarrow [B] \]
Semantics of Proof Terms

A classical axiom - excluded middle

\[ \vdash P \lor \neg P \quad \text{BY } Magic(P) \]

\[ Magic(P) \in \left[ P \lor \neg P \right] \]
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 $\text{Magic}(P)$ 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 $p(t_1, \ldots, t_n)$.

If $P,Q$ are problem statements (predicate formulas), then to assert $P \land Q$ means to find proofs or constructions $p$ and $q$ for $P$, $Q$ respectively.

$P \lor Q$ 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 \( p_a \) of \( P[a/x] \), taken together, \( <a, p_a> \) solves this problem or proves this formula.
Programming by refinement

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

$$\forall x : A. \exists y : B_1 \times B_2. R_1(x, \text{fst}(y)) \& R_2(x, \text{snd}(y))$$

We might decompose this goal into two subgoals

$$x : A \vdash \exists y : B_1 \times B_2. (R_1 \& R_2)$$

$$x : A \vdash \exists y : B_1. R_1$$

$$x : A \vdash \exists y : B_2. R_2$$
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 NatInd 1**

.... induction case.....

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

**BY existsR [0] 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 \[(r + 1)^2 \leq i\] THEN Auto**
Proof of Root Theorem (cont.)

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

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

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

\( \text{BY } \texttt{existsR} [r + 1] \ \text{THEN} \texttt{Auto} \)

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

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

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

\( \text{BY } \texttt{existsR} [r] \ \text{THEN} \texttt{Auto} \)
The Root Program Extract

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

```ml
let rec sqrt i =
    if i = 0 then < 0, pf0 >
    else let < r, pf_{i-1} >= sqrt(i - 1)
    in if (r + 1)^2 <= n then < r + 1, pf_i >
        else < r, pf_i' >
```
A Recursive Program for Integer Roots

Here is a very clean functional program

\[ r(n) := \begin{cases} 0 & \text{if } n = 0 \\ \text{let } r_0 = r(n-1) \text{ in} \\ \begin{cases} (r_0 + 1)^2 \leq n & \text{then } r_0 + 1 \\ r_0 & \text{else} \end{cases} \end{cases} \]

This program is close to a declarative mathematical description of roots given by the following theorem.
Theorem \( \forall n : \mathbb{N}. \exists r : \mathbb{N}. \text{Root} (r, n) \)

Proof by induction

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

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

Choose \( r_0 \) where \( \text{Root}(r_0, n-1) \), i.e. \( r_0^2 \leq n-1 < (r_0+1)^2 \)

\( (r_0 + 1)^2 \leq n \lor n < (r_0 + 1)^2 \)

case \( (r_0 + 1)^2 \leq n \) then \( r = (r_0 + 1) \)

\( (r_0 + 1)^2 \leq n < ((r_0 + 1)^2 < (r_0 + 2)^2) \)

case \( n < (r_0 + 1)^2 \) then \( r = r_0 \) since \( r_0 \leq n-1 < n \).

Qed
Deduction Systems

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

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

\[ \overline{H} \vdash G \quad \overline{H} \vdash \overline{G} \]

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

\[ n : \mathbb{N} \vdash \exists r : \mathbb{N}. R(n, r) \text{ BY } \text{existsR}(0, pf) \]

\[ n : \mathbb{N} \vdash R(n, 0) \text{ BY } pf \]

The evidence for \( \exists r : \mathbb{N}. R(n, r) \) should be a pair which is what the proof provides \( \langle 0, pf \rangle \) written \( \text{existsR}(0, pf) \)
Semantics of Proof Terms - Example

\[
n : \mathbb{N} \vdash \exists r : \mathbb{N}. R(n, r) \text{ BY } \text{existsR}(0, pf)
\]

\[
n : \mathbb{N} \vdash R(n, 0) \text{ BY } pf
\]

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

Example

\[ \overline{H} \vdash A \& B \text{ BY } \text{and} R\langle pfa, pfb \rangle \]
\[ \overline{H} \vdash A \text{ pfa} \]
\[ \overline{H} \vdash A \text{ pfb} \]

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

Example

\[ \overline{H} \vdash A \land B \ \text{BY and} \ R\langle pfa, pfb \rangle \]
\[ \overline{H} \vdash A \ \text{pfa} \]
\[ \overline{H} \vdash A \ \text{pfb} \]

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

\[ \overline{H} \vdash \forall x : A. R(x) \text{ BY all} R(x. pfb) \]

\[ \overline{H}, x : A \vdash R(x) \text{ BY } pfb \]
Semantics of Proof Terms

\[ \overline{H} \vdash \forall x : A. \ R(x) \ \text{BY all} \ R(x, pfb) \]

\[ \overline{H}, \ x : A \vdash \ R(x) \ \text{BY pfb} \]
Constructive Semantics

Notice that the proof term corresponding to

\[ \forall x : A. R \text{ is } \text{allR}(x. pf) \]

This should denote an element of \( x : \llbracket A \rrbracket \rightarrow \llbracket R_x \rrbracket \)

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
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 Interactive Program for Integer Roots

\[
\text{r := 0; } \\
\text{While } (r + 1)^2 \leq n \text{ do } \\
\quad \text{r := r + 1} \\
\text{od}
\]
An Interactive Program for Integer Roots, continued

\[
r := 0;
\]
\[
\textbf{While} (r + 1)^2 \leq n \textbf{ do}
\]
\[
r := r + 1
\]
\[
\textbf{od}
\]
\[
n < (r + 1)^2
\]
A Program for Integer Roots With Assertions

\[ r := 0; \quad r^2 \leq n \]

While \((r + 1)^2 \leq n\) do
\[ (r + 1)^2 \leq n \]
\[ r := r + 1 \]
\[ r^2 \leq n \]
end

\[ r^2 \leq n \]

\[ n < (r + 1)^2 \]

This program suggests a precise specification

\[ \text{Root} (r, n) \iff r^2 \leq n < (r+1)^2 \]

\[ r^2 \leq n \text{ is an invariant} \]
\[ \forall s : \{ n : \mathbb{N} ; r : \mathbb{N} \}. \exists s' : \{ n : \mathbb{N} ; r : \mathbb{N} \}. \ Root(s.n, s'.r) \]

The proof will build the while loop root finder and apply it to a state \( s' \) where \( s'.r = 0 \). Thus \( s'.r = \text{root}(s') \).
Lecture 2 - Outline

Semantics of Evidence
Constructive Semantics
Example
Semantics of Proof Objects
Imperative Realizers
Distributed Realizers
Computations with state: terminating, deterministic

\[ S_i : \{ n : \mathbb{N} ; r : \mathbb{N} \} \]

\[
\begin{array}{c|c}
S_0 & 17 \ 0 \\
S_1 & 17 \ 1 \\
S_2 & 17 \ 2 \\
\vdots \\
S_{17} & 17 \ 4 \\
\end{array}
\]
Computations with state: unbounded, deterministic

\[ S_i : \{ n : \mathbb{N} ; p : \{ x : \mathbb{N} \mid \text{Prime}(x) \} \} \]

\[
\begin{array}{c|c|c}
S_0 & 10 & 11 \\
S_1 & 10 & 13 \\
S_2 & 10 & 17 \\
\vdots & \vdots & \\
\end{array}
\]
Computations with state: reactive, nondeterministic

in: Input, S: State, out: Output

\langle [0], s_0, \text{nil} \rangle
\langle \text{nil}, s_1, [15] \rangle
\langle \text{nil}, s_2, [3] \rangle
\langle [1,5], s_3, [3,7] \rangle
\langle [5,8], s_4, \text{nil} \rangle
\cdots
Computations with state: asynchronous, distributed

- \( P_i \) 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 ⊕ B, where A, B are message automata
Message Automata – Guessing Roots

begin  \( x, n, r : \mathbb{N} \),  \( g : \mathbb{N} \) List,  \( rdy : \text{Bool} \)

initially  \( n = 2, r = 0, g = \text{nil}, rdy = \text{true} \)

send(\(<\text{out},<"\text{root}\?",n>\) )

[*collect guesses*]

input receive(\(<\text{In},<"\text{guess}\",x>>>;g := x\cdot g\)

[*check for a root among guesses so far*]

check : if rdy = true
then for all i on g do
if \( i^2 \leq n < (i + 1)^2 \)
then  \( r := i \)
rdy := false
\( g := \text{nil} \)
\( g := \text{nil} \)

send(\(<\text{out},<"\text{root}\",<n,r>>>\) )

exit
fi end
else skip fi

[*ask for another root*]

next: if rdy = false then \( n := n + 1; rdy := \text{true} \)
else skip fi

end
Picture of a Computation

\[ \text{loc}(1) \quad \text{loc}(2) \quad \text{loc}(3) \quad \text{loc}(4) \quad \text{loc}(5) \]

\[
\begin{align*}
<s_1,a_1,m_1>@t & \quad <s_2,a_2,m_2>@t & \quad <s_3,a_3,m_3>@t & \quad <s_4,a_4,m_4>@t & \quad <s_5,a_5,m_5>@t \\
\downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
<s_1,a_1,m_1>@t+1 & \quad <s_2,a_2,m_2>@t+1 & \quad <s_3,a_3,m_3>@t+1 & \quad <s_4,a_4,m_4>@t+1 & \quad <s_5,a_5,m_5>@t+1
\end{align*}
\]
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

Baire
Kleene
Gentzen
Heyting
Kronecker
Brouwer
Weyl
Baire
Borel
Bishop
Programming Logics

McCarthy
Lip, functional
Scott / Strachey
Milner LCE
Kahn,
Manna

Leibniz
Church
Brouwer
Kolmogorov
Martin-Löf

McCarthy
Scott
Milner
Kleene
(realizability)
RC
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 \(\text{Root}(n,r)\) has an element, the evidence for its truth
Propositions (cont.)

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

So \((x = x \text{ in } A)\) 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 \(x = x \text{ in } A\).)
### Propositions (cont.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Proposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ax} \in (a=b \ \text{in} \ A) )</td>
<td>– we know ( a ) equals ( b ) in ( A )</td>
</tr>
<tr>
<td>\text{empty} \hspace{1em} ( (0=1 \ \text{in} \ N) )</td>
<td>– there is no evidence that ( 0=1 )</td>
</tr>
<tr>
<td>( \text{ax} \in (0\neq1 \ \text{in} \ N) )</td>
<td>– there is evidence that ( 0 ) is not equal to ( 1 ) in ( N )</td>
</tr>
</tbody>
</table>

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.

\[
\text{root}(n) := \begin{cases} 
0 & \text{if } n = 0 \\
\text{let } r_0 = \text{root}(n/4) \text{ in} \\
\text{if } (2 \cdot r_0 + 1)^2 \leq n \\
\quad \text{then } 2 \cdot r_0 + 1 \\
\quad \text{else } 2 \cdot r_0 
\end{cases}
\]

since if \( n \neq 0, n/4 < n \)

This is an efficient recursive function, but why is it correct?
A Theorem that Roots Exist (Can be Found)

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

Pf by efficient induction

Base \( n = 0 \) let \( r = 0 \)

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

Choose \( r_0 \) where \( r_0^2 \leq n/4 < (r_0 + 1)^2 \)

note \( 4 \cdot r_0^2 \leq n < 4 \cdot (r_0 + 1)^2 = 4 \cdot r_0^2 + 8 \cdot r_0 + 4 \)

thus \( 2 \cdot r_0 \leq \text{root } (n) < 2 \cdot (r_0 + 1) \)

if \( (2 \cdot r_0 + 1)^2 \leq n \) then \( r = 2 \times r + 1 \)

since \( (2 \cdot r_0)^2 = 4 \cdot r_0^2 + 8 \cdot r_0 + 4 \)

else \( r = 2 \times r \) since \( (2 \cdot r_0)^2 \leq n < (2 \cdot r_0 + 1)^2 \)

Qed
Exercise

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

in:

\[
\begin{align*}
  r &:= 0; \quad r^2 \leq n \\
  \text{While} \quad (r + 1)^2 \leq n \quad \text{do} \\
  &\quad r := r + 1 \\
  \text{od} \\
  \text{Root}(n, r)
\end{align*}
\]
Lecture 3

A Theory of Events

Robert L. Constable
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
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*

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 \rightarrow E \rightarrow E + Loc
sender?: E \rightarrow 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  
\[ x = \text{pred}?(y) \text{ or } x = \text{sender}?(y). \]  
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
EVValue – events have values, val(e)
EState – locations have state, temporal operators
\textbf{initially, when, after}

ECom – communication topology is given by a graph
Ettime – 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 \rightarrow \text{Loc}
kind: E \rightarrow \text{KND}
first: E \rightarrow \text{B}
pred: e: \{x:E | \neg \text{first}(x)\} \rightarrow \{x:E | \text{loc}(x) = \text{loc}(e)\}
snder: \{x:E | \text{kind}(x) = \text{rcv}\} \rightarrow E
<: E \rightarrow E \rightarrow \text{B}
Ty: E \rightarrow \text{Type}
val: x:E \rightarrow Ty(x)
T: x:Id \rightarrow i:Loc \rightarrow \text{Type}
when: x: Id \rightarrow e: E \rightarrow T(\text{loc}(e), x)
after: x: Id \rightarrow e: E \rightarrow T(\text{loc}(e), x)
initial: x: Id \rightarrow i: Loc \rightarrow T(i, x)
Example – Two-Phase Handshake

\[
E_p = \{ e : E \mid \text{loc}(e) = p \}
\]

\[
\text{Snd}_{p,l} = \{ e : E_p \mid \text{sends}(e, l) \neq \text{nul} \}
\]

\[
\text{Rcv}_{p,l} = \{ e : E_p \mid \text{kind}(e) \text{ is receive on } l \}
\]
We illustrate this process by deriving a protocol for the two-phase handshake from a proof that its specification is realizable.

(1) \( \forall e_1, e_2 : \text{Snd}_{S, l_1}. \ \exists r : \text{Rcv}_{S, l_2}. \ (e_1 < e_2 \Rightarrow e_1 < r < e_2) \)

(2) \( \forall e_1, e_2 : \text{Snd}_{R, l_2}. \ \exists r_1, r_2 : \text{Rcv}_{R, l_1}. \ (e_1 < e_2 \Rightarrow r_1 < e_1 < r_2 < e_2) \)
Picture of a Computation

loc(1) \quad loc(2) \quad loc(3) \quad loc(4) \quad loc(5)

<s_1, a_1, m_1>@t \quad <s_2, a_2, m_2>@t \quad <s_3, a_3, m_3>@t \quad <s_4, a_4, m_4>@t \quad <s_5, a_5, m_5>@t

\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow

<s_1', a_1', m_1'>@t+1 \quad <s_2', a_2', m_2'>@t+1 \quad <s_3', a_3', m_3'>@t+1 \quad <s_4', a_4', m_4'>@t+1 \quad <s_5', a_5', m_5'>@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

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

\[ \begin{align*}
H_1 & \vdash G_1, \text{ ext } g_1 \\
H_2 & \vdash G_2, \text{ ext } g_2
\end{align*} \]
Refinements for Programs

\[ \vdash \forall x : A. \exists y : B. R(x, y) \quad \text{ext} \quad \lambda x. \text{cut}(x, z.g(x, z); l(x)) \]

\[ x : A \vdash \exists y : B. R(x, y) \quad \text{ext} \quad \text{cut}(x, z.g(x, z); l(x)) \]

by cut L

1. \[ x : A, z : L \vdash \exists y : B. R(x, y) \quad \text{ext} \quad g(x, z) \]

by D \[ g(x, z) \]

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

2. \[ x : A \vdash L \quad \text{ext} \quad l(x) \]

by ____

\[ \text{Refinements for Programs} \]

\[ \vdash \forall x : A. \exists y : B. R(x, y) \quad \text{ext} \quad \lambda x. \text{cut}(x, z.g(x, z); l(x)) \]

\[ x : A \vdash \exists y : B. R(x, y) \quad \text{ext} \quad \text{cut}(x, z.g(x, z); l(x)) \]

by cut L

1. \[ x : A, z : L \vdash \exists y : B. R(x, y) \quad \text{ext} \quad g(x, z) \]

by D \[ g(x, z) \]

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

2. \[ x : A \vdash L \quad \text{ext} \quad l(x) \]

by ____
Refinements for Systems

\[ \exists D : \text{System. } \forall es : \text{ES}(D). \]

\[ R(es) \text{ ext } \text{Comp}(\text{pf}_1(D_1, es_1), \text{pf}_2(D_2, es_2)) \]

by \text{Comp}

1. \( D_1 : \text{System}(G, \text{Loc}, \text{Lnk}) \)

\[ es_1 : \text{ES}(D_1) \vdash R_1(es_1) \text{ ext } \text{pf}_1(D_1, es_1) \]

2. \( D_2 : \text{System}(G, \text{Loc}, \text{Lnk}) \)

\[ es_2 : \text{ES}(D_2) \vdash R_2(es_2) \text{ ext } \text{pf}_2(D_2, es_2) \]
Two-Phase Handshake Theorem

Theorem:
\[ \forall e_1, e_2 : \text{Snd}_s . e_1 < e_2 \Rightarrow \exists r : \text{Rcv}_s . e_1 < r < e_2 \]

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 \( \ell \) 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 \( \ell' \).
Let \( r \) be this \( e' \).
If we have the lemmas, then the theorem is true.
Lemmas

L1. If $s$ sends on link $\ell$ then it waits
$$\forall e: Snds, \ell (rdy \text{ after } e) = \text{false}$$

L2. $s$ sends on link $\ell$ only when $rdy = \text{true}$
$$\forall e: Snds, \ell (rdy \text{ when } e) = \text{true}$$

L3. After $s$ on link $\ell$ sends it is ready only after an acknowledgement on link $\ell'$
$$\forall e : Snds, \ell e \text{ changes } rdy \text{ to true } \Rightarrow$$
$$e \text{ is a receive from } R \text{ on link } \ell'.$$
Realizing the Lemmas - 1

L1. $\forall e: Snds, \ell. (\text{rdy after } e) = false$
    We can require that the sends on $\ell$ are caused
    by the action of setting $\text{rdy}$ to false.
    a: $\text{rdy} := false; \text{send } (\ell, <\text{tag}, \text{m }>)$
    We also require that only action a can send on $\ell$.
    This is a frame condition.

L2. $\forall e: Snds, \ell. (\text{rdy when } e) = true$
    We require the precondition on the action a
    that $\text{rdy} = true$
    a: $\text{pre( rd y = true) } \Rightarrow \text{ rd y := false; send } (\ell, <\text{tag}, \text{m }>)$
Realizing the Lemmas - 2

L3: \( \forall e : SndS, \ell . (e \Delta \text{rdy}) = \text{true} \Rightarrow \)
\[ \text{kind}(e) = \text{rcv on } \ell' \]

We stipulate that a receive sets \text{rdy} to true
and that only a \text{rcv} or a \text{can} change \text{rdy}.

\[ b: \text{rcv}(\ell', m) \Rightarrow \text{rdy}:= \text{false} \]

only \([a,b]\) affect \text{rdy} (frame condition)
Handshake Message Automation

action local (a) sends on $l_1 < \text{tag, v }$
only [a] sends on $l_1$
state $\text{rdy} : \mathbb{B}$; initially $\text{rdy} = \text{true}$
precondition $a$ is $\text{rdy} = \text{true}$
effect local (a) $\text{rdy} := \text{false}$
action $\text{rcv}_{\ell_2} < \text{ack} > : \text{Atom}$
effect $\text{rdy} := \text{true}$
only [local (a), $\text{rcv}_{\ell_2} < \text{ack}>$] affect $\text{rdy}$
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 $n(i) = \text{dst(out}(i))$, 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

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

Theorem \( \forall R: \text{List}(\text{Loc}). \ \text{Ring}(R) \)
\( \exists D: \text{Dsys}(R). \ \text{Feasible}(D) \) &
\( \forall es: \text{ES}. \ \text{Consistent}(D, es). \ \text{Leader}(R, es) \)
Logically Decomposing the Leader Election Task

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

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

2. \( \forall e. \text{kind}(e) = \text{rcv}(\text{in}(i), <\text{vote},u>) \Rightarrow \) 
   \( (u > \text{uid}(i) \Rightarrow \exists e'. \text{kind}(e') = \text{rcv}(\text{out}(i), <\text{vote},u>)) \)

3. \( \forall e'. [ (\text{kind}(e') = \text{rcv}(\text{out}(i), <\text{vote},\text{uid}(i)>)) \lor \) 
   \( \exists e. (\text{kind}(e) = \text{rcv}(\text{in}(i), <\text{vote},u>) \& (e < e' \& u > \text{uid}(i))) ] \)

4. \( \forall e@i. \text{kind}(e) = \text{rcv}(\text{in}(i), \text{uid}(i)). \exists e'@i. \text{kind}(e') = \text{leader} \)

5. \( \forall e@i. \text{kind}(e) = \text{leader}. \exists e@i. \text{kind}(e) = \text{rcv}(\text{in}(i), <\text{vote},\text{uid}(i)>)) \)
Realizing Leader Election

Theorem
\[ \forall R:\text{List(Loc)}.\text{Ring}(R) \]
\[ \exists D:\text{Dsys}(R).\text{Feasible}(D). \]
\[ \forall es:\text{Consistent}(D, es). \left( \text{LE}(R, es) \Rightarrow \text{Leader}(R, es) \right) \]

Proof:
Let \( m = \max \{ \text{uid}(i) \mid i \in R \} \), then \( \text{ldr} = \text{uid}^{-1}(m) \).
We prove that \( \text{ldr} = \text{uid}^{-1}(m) \) 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 $u > \text{uid}(i)$ 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.

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

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

By induction on distance of i to ldr.

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

\((j = ldr \lor d(ldr, j) < d(ldr, i))\)

By induction on causal order of \( \text{rcv} \) events.

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

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

Hence, by Lemma 2 \( i = ldr \lor (d(ldr, i) < d(ldr, i)) \)

but the right disjunct is impossible.

Finally, from property 4, it is enough to know

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

which follows from Lemma 1.

QED
Realizing the clauses of LE(R,es)

We need to show that each clause of \( \text{LE}(R, es) \) 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. \( \exists e.\text{kind}(e) = \text{rcv}(\text{out}(i), <\text{vote, uid}(i)>) \)

We need to send \(<\text{vote, uid}(i)\>\) from each \(i\).

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

\(\forall e @ i. (\text{me when } e) = \text{uid}(i)\)

The Send Once Lemma lets the process at \(i\) send \(\text{uid}(i)\)

\(\exists e.\text{kind}(e) = \text{rcv}(\text{out}(i), <\text{vote, uid}(i)>)\)
Implementing Clauses 4, 5 of LE(R,es)

We can instantiate the Trigger Lemma to obtain

\[
\forall i: \text{Loc.} \\
\forall e' @ i.\text{kind}(e') = \text{leader}. \\
(\exists e @ i. e < e'.\text{kind}(e) = \text{rcv}(\text{in}(i), \langle \text{vote}, \text{uid}(i) \rangle)) \\
\forall e @ i.\text{kind}(e) = \text{rcv}(\text{in}(i), \langle \text{vote}, \text{uid}(i) \rangle) \Rightarrow \\
\exists e' @ i.\text{kind}(e') = \text{leader}
\]
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_\Delta e as x after e \neq x when e
@i only L affects x
realizes : \forall e@i. (x_\Delta e \Rightarrow \text{kind}(e) \in L)
Effect lemma

effect
@i state x:T₁; action k:T₂;
k(v) effect x: = f(s when e, val(e))

realizes
∀e@i. kind(e)=k  ⇒
x after e = f(s when e, val(e))
pre

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

∀e @i. (kind(e) = k ⇒ p(s where, val(e)))
& ∀e @i. ∃e' @i. e ≤ e' &
(kind(e') = k ∨ ∀v : T. ¬p(s after e', v))
& ∃v : T. p(init(es) (i), v) ⇒ ∃e : E. loc(e) = i

The skolem function in the ∀e ∃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
≡_{\text{def}} \text{rec}(X.\text{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{KndList})
+\text{IdLnk}\times \text{Id}\times (\text{KndList})
+\text{Id}\times ds:x:Id \text{ fp} \rightarrow \text{Type}\times \text{Knd}\times T:\text{Type}\times x:Id\times (\text{State}(ds) \rightarrow T \rightarrow \text{DeclaredType}(ds;x))
+ds:x:Id \text{ fp} \rightarrow \text{Type}
 \times \text{Knd}
 \times T:\text{Type}
 \times \text{IdLnk}
 \times dt:x:Id \text{ fp} \rightarrow \text{Type}
 \times ((tg:\text{Id}\times (\text{State}(ds) \rightarrow T \rightarrow (\text{DeclaredType}(dt;tg) \text{ List}))) \text{ List})
+\text{Id}\times ds:x:Id \text{ fp} \rightarrow \text{Type}\times \text{Id}\times T:\text{Type}\times (\text{State}(ds) \rightarrow T \rightarrow \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 $T_1 \subseteq T_2$ or $T_2 \subseteq T_1$
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

\[
\text{\textit{\( \exists D : \text{DSyst. Feasible}(D) \& \forall es:ES. \\text{Consistent}(D, es) \Rightarrow P(es) \)\)}}
\]

We say that \textit{Feasible}(D) if D has at least one execution.

We say \textit{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.
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.

$$\text{Consistent}(D, es)$$

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) ⇒ ∃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(D, es(w)).
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 intuitions
- integrated into LPE, hence into procedural programming
- completely formalized
- supports proofs-as-processes of synthesis and programming
- widely applicable: verification, optimization, 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

- \(@i x : T \text{ initially} = v\)
- \(@i \text{ effect} k(v : t) \text{ on} x\)
  
  \[ x := f \text{ state} f \]
- \(@i \text{ precondition} a(v:t) \text{ is} P \text{ state} v\)
- \(k(v : T) \text{ sends on link} l\)
  
  \[[tg_1, f_1 \text{ state} v; \ldots; tg_n, f_n \text{ state} v]\]
- \(@i \text{ only} [k_1, \ldots, k_n] \text{ sends on link} l \text{ with tag}tg\)
- \(A \oplus B, \text{ where} A, B \text{ are message automata}\)
Citation

On urelements in computation
Andreas Blass, Yuri Gurevich, and Saharori Shelah, Choiceless Polynomial Time,
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
There are two kinds of security models

<table>
<thead>
<tr>
<th>Analytic</th>
<th>Algebraic</th>
</tr>
</thead>
<tbody>
<tr>
<td>learning ≠ Acquiring</td>
<td>to learn is to acquire</td>
</tr>
<tr>
<td>general computation</td>
<td>restricted computation</td>
</tr>
</tbody>
</table>
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.
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.
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.

<table>
<thead>
<tr>
<th>Sets</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Urelements</td>
<td>Atoms</td>
</tr>
</tbody>
</table>
On urelements in computation
Properties of Atoms

- Atoms are not enumerable, not infinite. Yet they are not of a fixed finite cardinality.
- The canonical elements are \textit{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 \textit{supervaluation} semantics).
- $\textit{token\{i:ut\}}$ 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, \( J( a, b, c, \ldots ) \), is true for any permutation \( a \) to \( a^{1} \), \( b \) to \( b^{1} \), \( \ldots \), i.e. \( J(a^{1}, b^{1}, c^{1}, \ldots ) \).
Examples

\[ f(x) = \begin{cases} \text{token}(a) & \text{if } x = 1 \\ \text{token}(b) & \text{else} \end{cases} \]

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

\[ f\{a,b\}(x) = \ldots \] would be legal.

It is illegal to introduce a term, say \textit{oups}, and have a rule \textit{oups} \downarrow \text{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
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)

\[
H \vdash (t : T \parallel a) \\
H \vdash t \in T \\
H \vdash a \in \text{Atom} \\
t \text{ is closed and mentions no tokens}
\]
Independence Rules

Base (\( t \) is a different atom)

\[
H \vdash (t: \text{Atom} \parallel a)
\]

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

Application (key rule)

\[
H \vdash (f(t):B[t/v] \parallel a)
\]

\[
H \vdash (f:(v:A \rightarrow B) \parallel a)
\]

\[
H \vdash (t:A \parallel a)
\]
Independence Rules

Absurdity \((a \text{ depends on } a)\)

\[ H \vdash \neg (a:Atom \parallel a) \]

Set (separating predicate is irrelevant)

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

\[ H \vdash (t:T \parallel a) \]

\[ H \vdash t \text{ in } \{x:T \mid P\} \]
Independence Rules

Equality

\[ H \vdash (t_1 : T_1 \parallel a_1) = (t_2 : T_2 \parallel a_2) \]

\[ H \vdash T_1 = T_2 \text{ in Type} \]

\[ H \vdash t_1 = t_2 \text{ in } T \]

\[ H \vdash a_1 = a_2 \text{ in } Atom \]

This allows us to build up independence from pieces.
Basic Facts

We can prove general facts such as

**Theorem** $\forall a : Atom. \forall z : \mathbb{Z} . (z : \mathbb{Z} \parallel a)$

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 $(z : \mathbb{Z} \parallel a)$, then $(z + 1 : \mathbb{Z} \parallel a)$ since $z + 1$ is

$\lambda(x.x + 1)(z)$ and $\lambda(x.x + 1)$

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 $P_{\text{crypto}}$ and the security property with $Q_{\text{secure}}$. Then we show that some message automaton $M$ realizes $P_{\text{crypto}} \implies Q_{\text{secure}}$.

$$\vdash (P_{\text{crypto}} \implies Q_{\text{secure}}) \text{ extract } M$$
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 $L$.

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

$$Q_1(a) == \forall e. [\text{loc}(e) \in L \lor ((\text{state}(\text{loc}(e)))\text{when e}) \parallel a)]$$

The liveness part is that eventually all agents on $L$ receive the secret.

$$Q_2(a) == \forall i. i \in L \Rightarrow \exists e. (\text{loc}(e) = i \land (x \text{ when e}) = a)$$

The full specification is

$$\exists a : \text{Atom} \cdot Q_1(a) \land Q_2(a)$$
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

\[ \forall i, j : \text{Loc. } i \neq j \Rightarrow \forall a : \text{Atom. } a \in (\text{Nonces initially } i) \Rightarrow (j \parallel a) \]
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.
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

<table>
<thead>
<tr>
<th>Models</th>
<th>Code</th>
<th>People</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-line banking model (logic book)</td>
<td>Secure aps (thru provers) banking sys</td>
<td>Aps programmer</td>
</tr>
<tr>
<td>Protocols IOA, MA ASM</td>
<td>Middleware</td>
<td>Aps theory (transaction design)</td>
</tr>
<tr>
<td>Computable function models</td>
<td>Prog languages procedural, declarative</td>
<td>Syst. Prog. Distrib sys theory</td>
</tr>
<tr>
<td>Concurrency, file syst</td>
<td>Operating system Virtual machines</td>
<td>Computer writer computer theory</td>
</tr>
<tr>
<td>Modeling and simulation languages</td>
<td>Assembler Handware</td>
<td>Semantics</td>
</tr>
</tbody>
</table>

This is how we span levels of abstraction
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.