

A Logic of Events, Proofs-as-Processes, and Conclusion

Robert Constable

Cornell University

Outline

Review Propositions-as-types, Proof terms, Proofs-as-Programs
Discuss new types and records

Challenges for proofs-as-processes

Logic of Events and specifications of tasks

Message automata and IO-automata

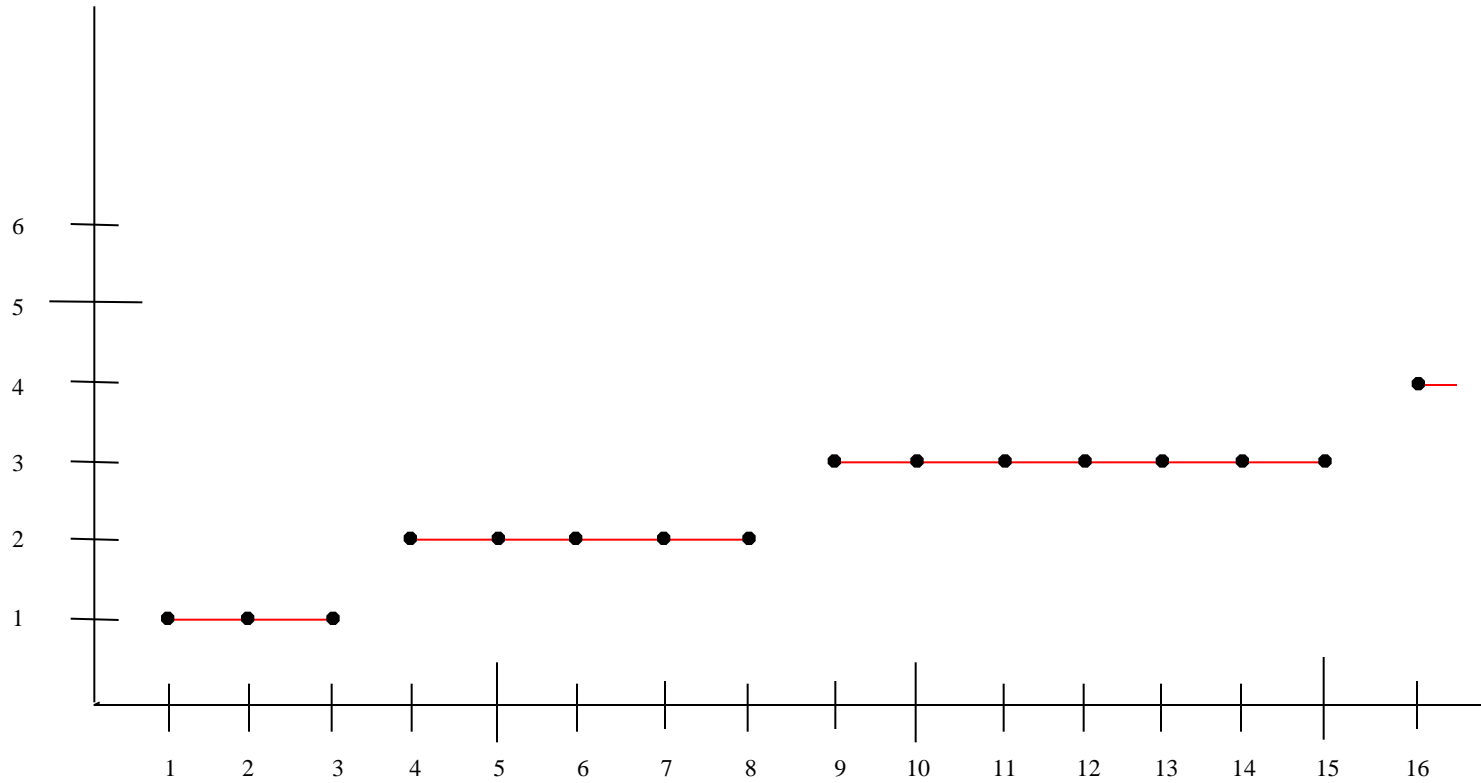
example protocol and proof of correctness

Event classes and **synthesis of processes**

Fault-tolerant and attack-tolerant systems

Conclusion for the course

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 ≤ i]` THEN `Auto`

Proof of Root Theorem (cont.)

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

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

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

BY `existsR [r + 1]` THEN `Auto'`

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

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

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

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:

```
let rec sqrt i =  
  if i = 0 then < 0, pf0 >  
  else let < r, pfi-1 > = sqrt i - 1  
  in if r + 12 ≤ n then < r + 1, pfi >  
  else < r, pfi' >
```

A Recursive Program for Integer Roots

Here is a very clean **functional program**

```
r(n):= if n= 0 then 0
      else let r0 = r (n-1) in
      if (r0 + 1)2 ≤ n then r0 + 1
      else r0 fi
      fi
```

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

A Program for Integer Roots With Assertions

```
r := 0; r2 ≤ n
  While (r + 1)2 ≤ n do
    (r + 1)2 ≤ n
    r := r + 1
    r2 ≤ n
  od
r2 ≤ n
n < (r + 1)2
```

This program suggests the precise specification

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

$r^2 \leq n$ is an **invariant**

An Efficient Extract

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

Pf by **efficient induction**

Base $n = 0$ let $r = 0$

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

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 $2 \cdot r_0 \leq \mathbf{root} (n) < 2 \cdot (r_0 + 1)$

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

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

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

Qed

Efficient Root Program

```
root(n) := if n=0 then 0
else let r0 = root (n/4) in
if (2·r0+1)2 ≤ n
    then 2·r0+1
    else 2·r0 fi
fi
since if n ≠ 0, n/4 < n
```

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

Correctness of the Recursive Program

Use this “efficient induction principle”.

We can **implement the principle** by proving with ordinary induction that an **efficient realizer** (built with the Y combinator) belongs to this type.

$$P(0) \ \& \ \forall n: \mathbb{N}.(P(n/4) \Rightarrow P(n)) \Rightarrow \forall n: \mathbb{N}.P(n)$$

General Recursion in CTT

$f(x) = F(f,x)$ is a recursive definition, also
 $f = \lambda(x.F(f,x))$ is another expression of it, and the
CTT definition is:

$$\text{fix}(\lambda(f. \lambda(x. F(f,x)))$$

which reduces in one step to:

$$\lambda(x.F(\text{fix}(\lambda(f. \lambda(x. F(f,x))))),x))$$

by substituting the **fix term** for f in $\lambda(x.F(f,x))$.

Non-terminating Computations

CTT defines **all general recursive functions**,
hence non-terminating ones such as this

$$\text{fix}(\lambda(x.x))$$

which in one reduction step **reduces to itself!**

This system of computation in the object language is a simple **functional programming language**.

Subtyping and Polymorphism

There is a primitive **subtyping** relation in CTT.

$A \sqsubseteq B$ means that the elements of A are elements of B and **$a=b$ in A implies $a=b$ in B** . Here are some basic facts about subtyping:

$$x : Z \mid x > 0 \sqsubseteq Z$$

$$(A \sqsubseteq A' \ \& \ B \sqsubseteq B') \Rightarrow A \times B \sqsubseteq A' \times B'$$

$$(A \sqsubseteq A' \ \& \ B \sqsubseteq B') \Rightarrow A + B \sqsubseteq A' + B'$$

$$(A \sqsubseteq A' \ \& \ B \sqsubseteq B') \Rightarrow A' \rightarrow B \sqsubseteq A \rightarrow B'$$

Record Types and Inheritance

We can define algebraic structures as records. For example, a monoid on carrier S is a record type over S with two components, an associative operator and an identity:

$$\mathit{Monoid} = \{ \mathit{op}: S \times S \rightarrow S; \mathit{id}: S \}$$

A group extends this record type on S by including an inverse operation.

$$\mathit{Group} = \{ \mathit{op}: S \times S \rightarrow S; \mathit{id}: S; \mathit{inv}: S \rightarrow S \}$$

A *Group* is a subtype of a *Monoid* as we show next.

$$\mathit{Group} \sqsubseteq \mathit{Monoid}$$

Groups and Monoids as Records

The basic idea is that the elements of a record type are functions from the field selectors names, e.g. $\{op, id, inv\}$ to elements of types assigned to them by a mapping called a signature, $Sig:\{op, id, inv\} \rightarrow Type$. Here are the mappings for a Group over the carrier S .

$$Sig(op) = S \times S \rightarrow S, \quad Sig(id) = S, \quad Sig(inv) = S \rightarrow S$$

Monoid and Group are these dependent function spaces, Group a subtype of Monoid.

$$i: op, id, inv \rightarrow Sig(i) \sqsubseteq i: op, id \rightarrow Sig(i)$$

Inheritance among Algebraic Structures

Notice that the subtyping of algebraic structures depends on the function space subtyping, namely

$$i: op, id, inv \rightarrow Sig(i) \sqsubseteq i: op, id \rightarrow Sig(i)$$

is an instance of the general relation:

$$(A \sqsubseteq A' \ \& \ B \sqsubseteq B') \Rightarrow A' \rightarrow B \sqsubseteq A \rightarrow B'$$

Intersection and Top Types

We can build records using a binary **intersection** of types, $A \cap B$

These are the elements in both types A and B with $x=y$ in the intersection iff $x=y$ in A & $x=y$ in B.

Top is the type of **all closed terms** with the trivial equality, $x=y$ for all x, y in Top. Note for any type A, we have $A \sqsubseteq Top$ and $A \cap Top = A$.

Building Records by Intersection

Record types can be built by intersecting singleton records as follows. Let

$Id = \{x, y, z, \dots\}$ and $Sig: Id \rightarrow Type$ where

$Sig(i) = Top$ as the default. Then

$$x:A \Pi y:B = \begin{cases} \{x:A ; y:B\} & \text{if } x \neq y \\ \{x:A \Pi B\} & \text{if } x = y. \end{cases}$$

See [Kopylov PhD thesis](#), 2003.

Axiomatizing Co-inductive Types

In 1988 before we added intersection types to CTT, we axiomatized co-inductive types and implemented them in Nuprl as primitive.

Now with intersection types and the Top type, we can define them.

See Bickford, Constable, Gauspari 2010.

Defining Co-recursive Types in CTT

Let F be a function from types to types such as $F(T) = \mathbf{N} \times T$ or $F(T) = \text{St} \rightarrow \text{In} \rightarrow \text{St} \times T$. Define objects of the co-recursive type $\text{corec}(T, F(T))$ as the intersection of the iterates of F applied to Top .

$$\bigcap_{n:\mathbf{N}} F^n(\text{Top})$$

To build elements, we take the fixed point of a function f in the following type.

$$\bigcap_{T:\text{Type}} T \rightarrow F(T)$$

Elements of Co-inductive Types

For example to build elements of the co-recursive type for the function $F(T)$ given by

$$St \rightarrow In \rightarrow St \times T$$

we use $\text{fix}(\lambda(t.\lambda(s,i.<\text{update}(s,i),t>)))$.

It is easy to show by induction that this belongs to the co-recursive type. If the function F is continuous, the type is a fixed point of F , $F(\text{corec}(T.F(T))) \equiv \text{corec}(T.F(T))$.

Outline

Challenges for Proofs-as-Processes

What is the right **model of processes**? Can they be modeled as terms?

How to specify distributed computing tasks?

What are the right proof rules?

Can we synthesize real code?

The Story

We started by using IOA as our internal model of processes and a distributed database under our proof assistant. In 2003 we modified IOA to **Message Automata** and built an **event logic** around this model. These MA used **frame conditions** to render composition as union.

Year by year as we tackled harder protocols, we have been forced to be **more and more abstract** in order to complete the proofs and extract protocols, and we are being forced to replicate the database.

The Story continued

Now we can create **a variety of protocols** from proofs, e.g. consensus (e.g. Paxos, 2/3), authentication, group membership, etc.

We found advantages of starting very abstractly, e.g. we can generate many provably correct variants at the same time, providing **attack-tolerance**.

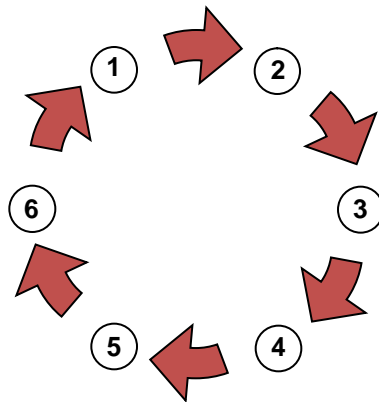
The Story continued

Our constructive proofs of consensus require proofs of **non-blocking**. I discovered that **FLP** can be proved constructively for effectively non-blocking protocols.

From **Constructive FLP** we can build an unbeatable adversary (attacker) against deterministic consensus.

Specification for Leader Election in a Ring

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



Let $n(i) = \text{dst}(\text{out}(i))$, 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) == $\exists \text{ldr: R. } \exists e@ \text{ldr. kind}(e)=\text{leader} \ \&$
 $\forall i:R. \forall e@i. \text{kind}(e)=\text{leader} \Rightarrow i=\text{ldr}$

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

Decomposing the Leader Election Task

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

1. $\exists e. \text{kind}(e)=\text{rcv}(\text{out}(i), \langle \text{vote}, \text{uid}(i) \rangle)$

2. $\forall e'. \text{kind}(e)=\text{rcv}(\text{in}(i), \langle \text{vote}, u \rangle) \Rightarrow$

$u > \text{uid}(i) \Rightarrow \exists e'. \text{kind}(e')=\text{rcv}(\text{out}(i), \langle \text{vote}, u \rangle)$

3. $\forall e'. \left(\text{kind}(e')=\text{rcv}(\text{out}(i), \langle \text{vote}, \text{uid}(i) \rangle) \right) \vee$

$\exists e. \left(\text{kind}(e)=\text{rcv}(\text{in}(i), \langle \text{vote}, u \rangle) \& \left(e < e' \& u > \text{uid}(i) \right) \right) \Big]$

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

5. $\forall e@i. \text{kind}(e)=\text{leader}. \exists e@i. \text{kind}(e)=\text{rcv}(\text{in}(i), \langle \text{vote}, \text{uid}(i) \rangle) \Big]$

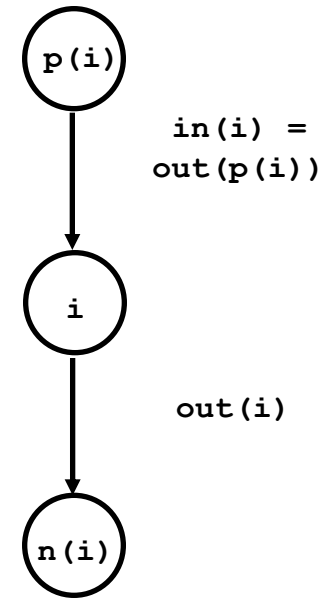
Realizing Leader Election

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

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 $u > uid(i)$ it sends it on.
3. Every rcv is either vote of predecessor $rcv_{in(i)}$ for itself or a vote larger than process id before.
4. If a process gets a vote for itself, it declares itself ldr.
5. If a processor declares ldr it got a vote for itself.



Lemmas

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

By **induction on distance of i to ldr** .

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

$(i = \text{ldr} \vee d(\text{ldr}, j) < d(\text{ldr}, i))$

By **induction on causal order of rcv events**.

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

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

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

but the right disjunct is impossible.

Finally, from property 4, it is enough to know

$\exists e. \text{kind}(e) = \text{rcv } (\text{in}(\text{ldr}), \langle \text{vote}, \text{uid}(\text{ldr}) \rangle)$

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

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 $out(i), vote$

Refinements for Systems

$\vdash \exists D : \text{System}. \forall es : \text{ES } D .$

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

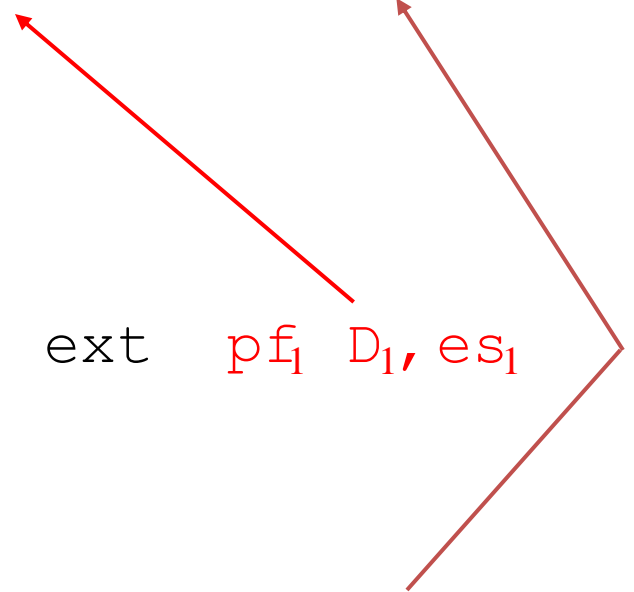
by Comp

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

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

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

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



Consensus is a Good Example

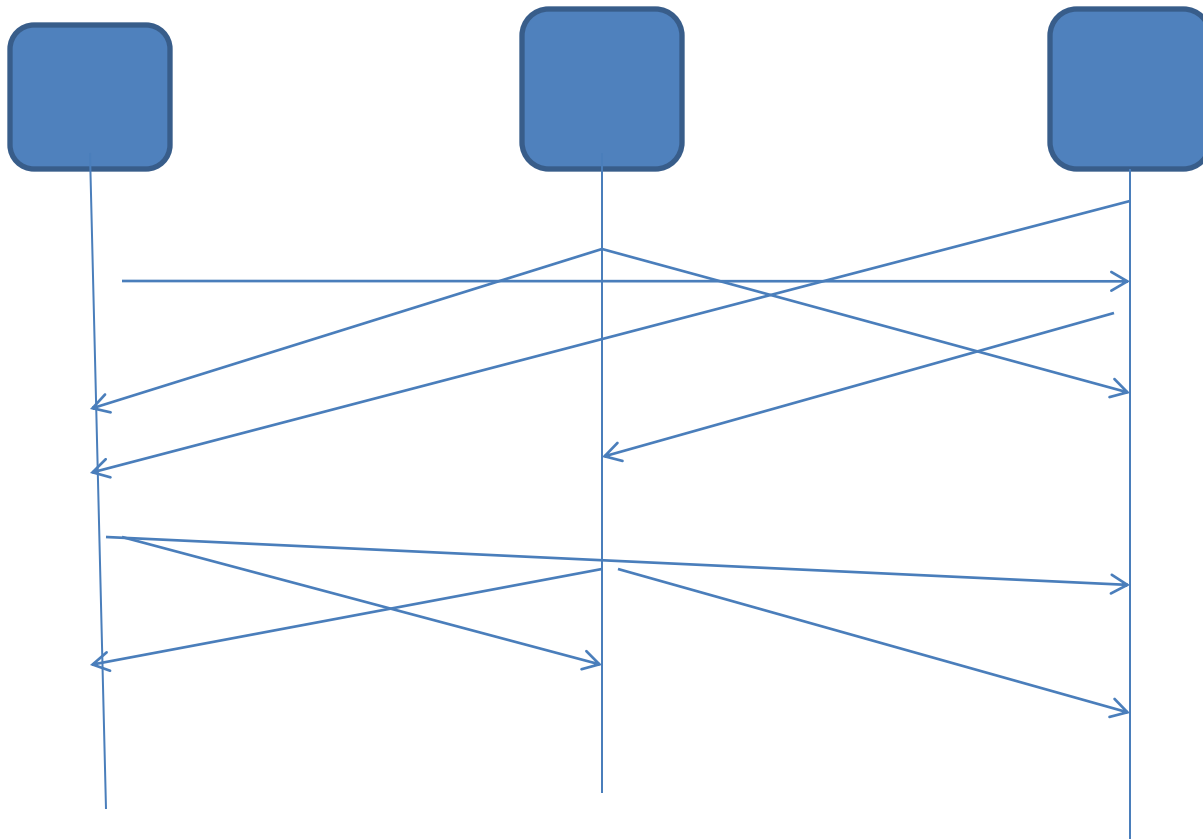
In modern distributed systems, e.g. the Google file system, clouds, etc., reliability against **faults** (crashes, attacks) is achieved **by replication**.



Consensus is used to coordinate write actions to keep the replicas identical. It is a **critical protocol** in modern systems used by IBM, Google, Microsoft, Amazon, EMC, etc.

Requirements of Consensus Task

Use **asynchronous** message passing to decide on a value.



Logical Properties of Consensus

P1: If all inputs are **unanimous** with value v , then any decision must have value v .

All $v:T$. (If All $e:E(\text{Input})$. $\text{Input}(e) = v$ then
All $e:E(\text{Decide})$. $\text{Decide}(e) = v$)

Input and **Decide** are **event classes** that effectively partition the events and assign values to them. The **events** are points in abstract space/time at which “information flows.” More about this just below.

Logical Properties continued

P2: All decided values are input values.

All $e:E(\text{Decide})$. Exists $e':E(\text{Input})$.

$e' < e$ & $\text{Decide}(e) = \text{Input}(e')$

We can see that P2 will imply P1, so we take P2 as part of the requirements.

Event Classes

If X is an **event class**, then $E(X)$ are the events in that class. Note $E(X)$ **effectively** partitions all events E into $E(X)$ and $E - E(X)$, its complement.

Every event in $E(X)$ has a value of some type T which is denoted **$X(e)$** . In the case of $E(\text{Input})$ the value is the typed input, and for $E(\text{Decide})$ the value is the one decided.

Events

Formally the type E of events is defined relative to the computation model which includes a definition of **processes**.

The events are the **points of space/time** at which information is exchanged. The information at an event e is **info(e)**.

Further Requirements for Consensus

The key **safety property** of consensus is that all decisions agree.

P3: Any two decisions have the same value.

This is called **agreement**.

All $e_1, e_2: E(\text{Decide})$. $\text{Decide}(e_1) = \text{Decide}(e_2)$.

Specific Approaches to Consensus

Many consensus protocols proceed in **rounds**, **voting on values**, trying to reach agreement. We have synthesized two families of consensus protocols, the **2/3 Protocol** and the **Paxos Protocol** families.

We structure specifications around **events during the voting process**, defining **E(Vote)** whose values are pairs $\langle n, v \rangle$, a **ballot number**, n , and a **value**, v .

Properties of Voting

Suppose a group G of n processes, P_i , decide by voting. If each P_i collects all n votes into a list L , and applies some **deterministic function $f(L)$** , such as majority value or maximum value, etc., then **consensus is trivial in one step**, and the value is known at each process in the first round – possibly at very different times.

The problem is much harder because of **possible failures**.





Fault Tolerance

Replication is used to ensure system availability in the presence of **faults**. Suppose that we assume that up to **f** processes in a group G of **n** might fail, then how do the processes reach consensus?

The **TwoThirds method** of consensus is to take $n = 3f + 1$ and **collect only $2f + 1$** votes on each round, assuming that **f** processes might have failed.

Example for $f = 1, n = 4$

Here is a sample of voting in the case $T = \{0,1\}$.

0	0	1	1	inputs
				
0_11	_011	001_	00_1	collected votes
1	1	0	0	next vote

00_1	001_	0_11	_011
0	0	1	1

where f is majority voting, first vote is input

Specifying the 2/3 Method

We can specify the fault tolerant 2/3 method by introducing further event classes.

$E(\text{Vote})$, $E(\text{Collect})$, $E(\text{Decide})$

$E(\text{Vote})$: the initial vote is the $\langle 0, \text{input value} \rangle$,
subsequent votes are $\langle n, f(L) \rangle$

$E(\text{Collect})$: collect $2f+1$ values from G into list L

$E(\text{Decide})$: **decide** v if all collected values are v

The Hard Bits

The small example shows what can go wrong with $2/3$. It can **waffle forever** between 0 and 1, thus never decide.

Clearly if there is a decide event, the values agree and that unique value is an input.





Can we say anything about eventually deciding, e.g. **liveness**?

Liveness

If f processes eventually fail, then our design will work because if f have all failed by round r , then at round $r+1$, all alive processes will see the same $2f+1$ values in the list L , and thus they will all vote for $v' = f(L)$, so in round $r+2$ the values will be unanimous which will trigger a decide event.

Example for $f = 1, n = 4$

Here is a sample of voting in the case $T = \{0,1\}$.

0	0	1	1	inputs
				
0 01_	001_	001_	_011	collected votes
0	0	0	1	next vote

000_	00_1	0_01	_001
0	0	0	0

where **f is majority voting**, first vote is input, round numbers omitted.

Safety Example

We can see in the $f = 1$ example that once a process P_i receives $2/3$ unanimous values, say 0, it is not possible for another process to overturn the majority decision.

Indeed this is a general property of a $2/3$ majority, the remaining $1/3$ cannot overturn it even if they band together on every vote.

Safety Continued

In the general case when voting is not by majority but using $f(L)$ and the type of values is discrete, we know that if any process P_i sees unanimous value v in L , then any other process P_j seeing a unanimous value v' will see the same value, i.e. $v = v'$ because the two lists, L_i and L_j at round r must share a value, that is they intersect.

Synthesizing the 2/3 Protocol from a Proof of Design

We can formally prove the safety and liveness conditions from the event logic specification given earlier.

From this **formal proof of design, pf**, we can automatically extract a protocol, first as an abstract process, then by verified compilation, a program in Java or Erlang.

The Synthesized 2/3 Protocol

Begin $r:\text{Nat}$, decided_i , $\text{vote}_i: \text{Bool}$,
 $r = 0$, $\text{decided}_i = \text{false}$, $v_i = \text{input to } P_i$; $\text{vote}_i = v_i$

Until decided_i **do**:

1. $r := r+1$
2. **Broadcast** $\langle r, \text{vote}_i \rangle$ to group G
3. **Collect** $2f+1$ round r votes in list L
4. $\text{vote}_i := \text{majority}(L)$
5. **If** $\text{unanimous}(L)$ **then** $\text{decided}_i := \text{true}$

End

General Process Model (GPM)

$M(P) == (\text{Atom List}) \times (T + P)$

$E(P) == (\text{Loc} \times M(P)) \text{ List}$

$F(P) = M(P) \rightarrow (P \times E(P))$

It is easy to show that M and E are continuous type functions and that F is weakly continuous. Thus for

$\text{Process} == \text{corec}(P. F(P))$

$\text{Msg} == M(\text{Process})$ and $\text{Ext} == E(\text{Process})$

we conclude Process is a subtype of $F(\text{Process})$,

$\text{Process} \subseteq \text{Msg} \rightarrow \text{Process} \times \text{Ext}$

Executing Systems of Processes

The **environment** chooses which messages will be delivered. A **run** of a system is an unbounded sequence of pairs $\langle \text{sys}, \text{choice} \rangle$.

From a run of a system, we can build **event structures** with locations and causal order.

Event Orderings over Runs

An event ordering of a run R is a collection of events E , a function loc giving the location of the event, a well founded $causal\ order\ <$ on events, and $info$, the information conveyed by an event: $\langle E, loc, <, info \rangle$

The $events$ are pairs $\langle x, n \rangle$ at which location x receives a message at step n of the run.

Event Structures over Runs

Event structures include the operations

x **when** e and x **after** e

for state variable x an events e , and the axiom

$\text{not first}(e)$ implies $(x \text{ when } e = x \text{ after pred}(e))$

Diversity

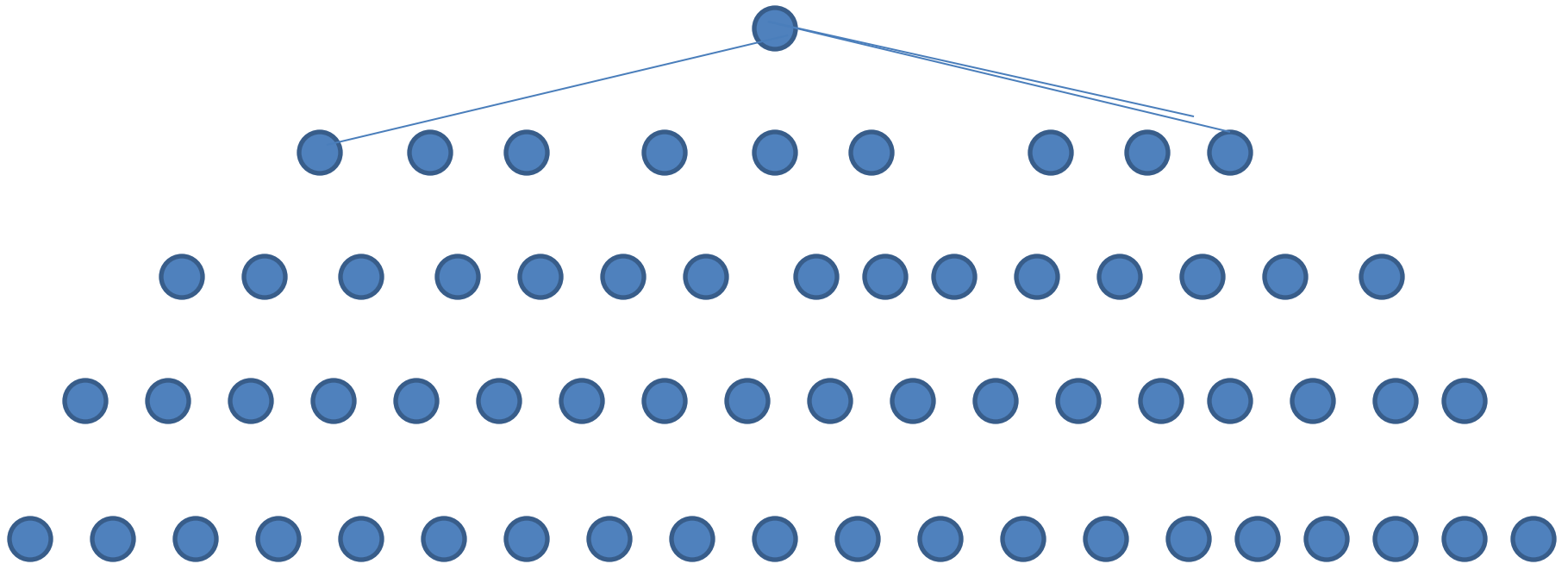
When we prove properties of a design, there are many options at several steps, and we are able to create **multiple proofs** at low additional cost. In the process **we create new designs**.

For example, for the 2/3 protocol, Mark Bickford found a variant that is faster by varying the design proof, as mentioned in our paper – he **varies the collection method**.

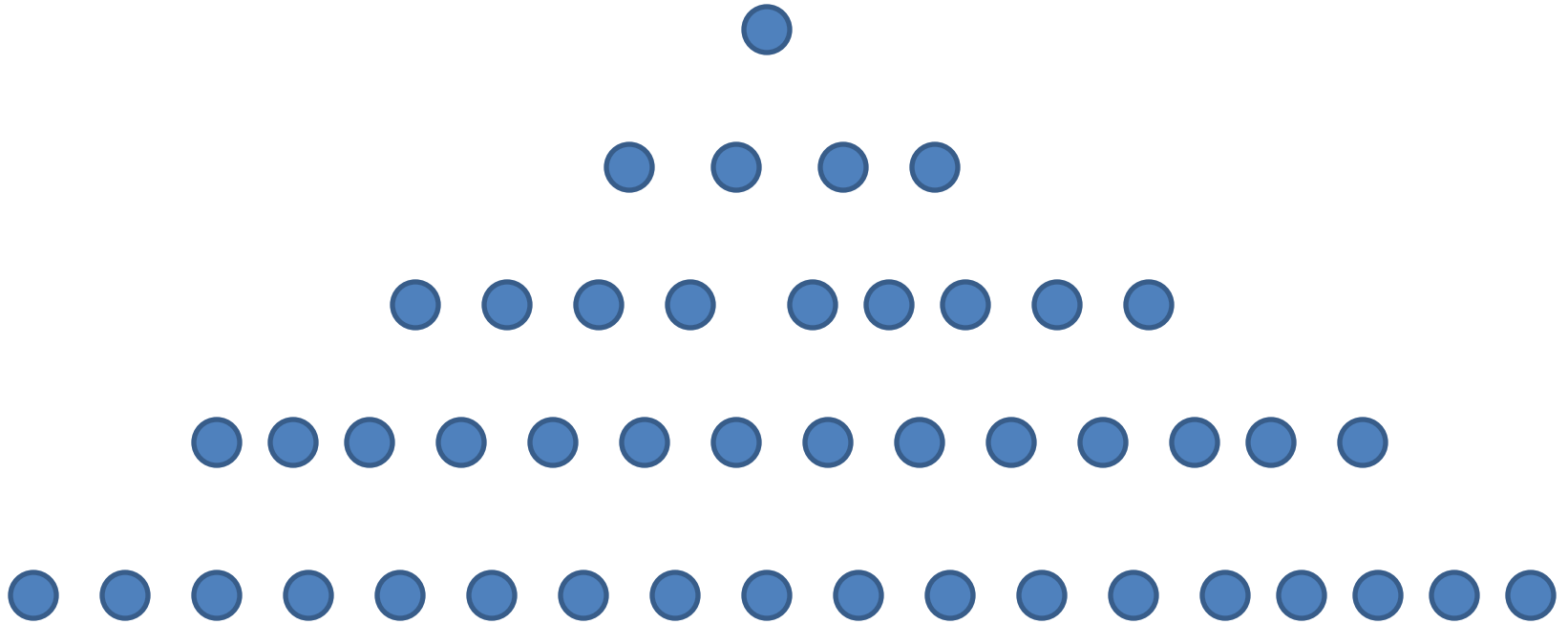
Diversity at the Level of Proof

Multiple **formal proofs** are “simultaneously” generated. We illustrate this by viewing a proof as a tree generated top down.

Illustrating Multiple Proofs



Illustrating Multiple Proofs



P1

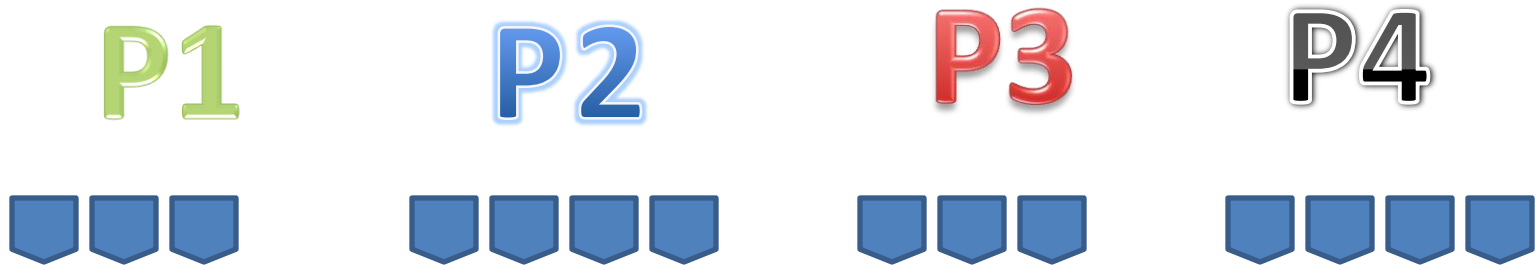
P2

P3

P4

Data Structure Diversity

Assuming there are four abstract protocols derived from the proof trees. For each of them it is possible to implement with different data structures, e.g. list, array, tree, set, etc.



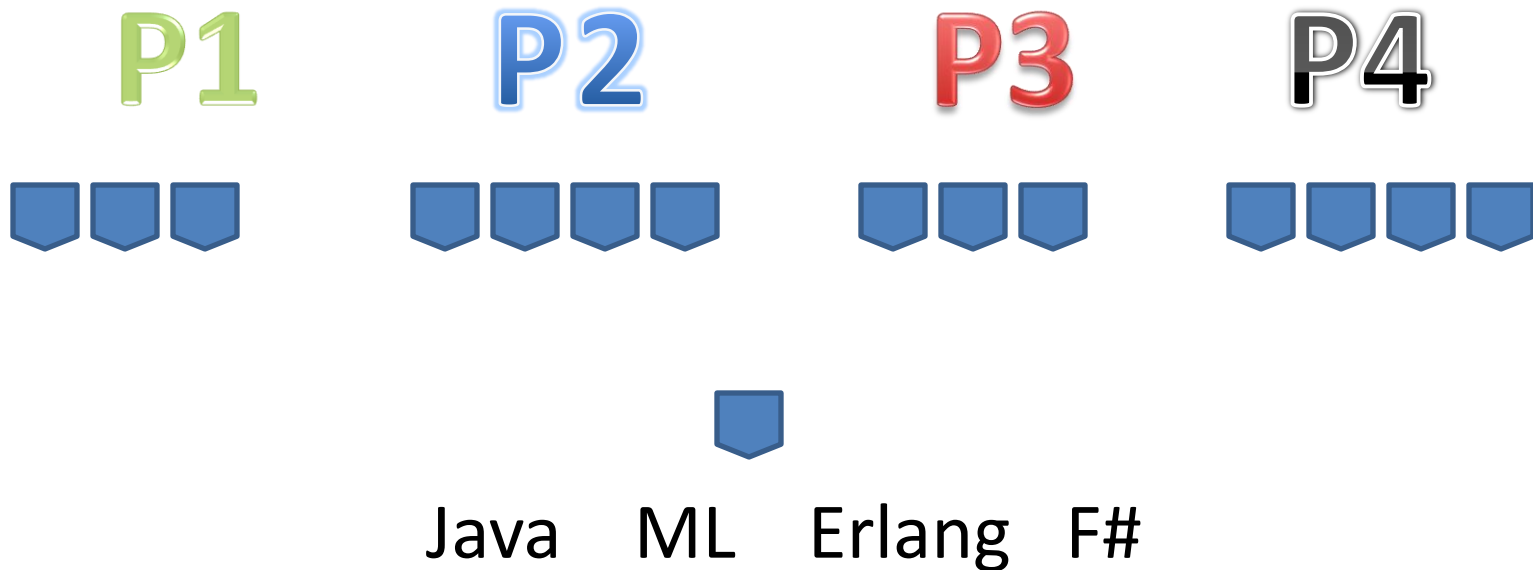
Programming Language Diversity

We can translate abstract programs into common programming languages such as Java, Erlang, C++, OCaml, or F#. So far we use only Java and Erlang.

Combining all levels of diversity we are able in principle to generate over **200 variants** of a protocol in the best case.

See Constable, Bickford, van Renesse 2010

Language Diversity



4 protocols, 14 options in 4 languages,
offers over **200 variants**

Role of the Environment

All distributed computing models must have a component that determines when messages between processes are delivered. We call this the **environment**. It introduces **uncertainty** into the model and determines the **schedule** of events.

A Fundamental Theorem of about the Environment

The **Fischer/Lynch/Paterson** theorem (**FLP85**) about the computing environment says:

it is not possible to deterministically guarantee consensus among two or more processes when one of them might fail.

We have seen the possibility of this with the 2/3 Protocol which could waffle between choosing 0 or 1. The environment can act as an adversary to consensus by managing message delivery.

The Environment as Adversary

In the setting of synthesizing protocols, I have shown that the FLP result can be made constructive (**CFLP**). This means that there is an algorithm, **env**, which given a potential consensus protocol P and a proof pf that it is **nonblocking** can create message ordering and a computation based on it, **env(P, pf)**, in which P runs forever, failing to achieve consensus.

Perfect Attacker

The algorithm $\text{env}(P, pf)$ is the perfect “denial of service attacker” against any consensus protocol P that is sensible (won't block).

Note, 2/3 will **block** if it waits for n replies or if it refuses to change votes as rounds progress.

Defending Against the Perfect Attack

One way to defend against $evn(P, pf)$ is to switch to another protocol P' if there appears to be an attack against P .

Definitions

P is called effectively **nonblocking** if from any reachable global state s of an execution of P and any subset Q of $n - t$ non-failed processes, we can find an execution from s using Q and a process P in Q which decides a value v .

Constructively this means that we have a computable function, **$wt(s, Q)$** which produces an execution and a state s in which a process, say P decides a value v .

Constructive FLP

Theorem (Constructive FLP): Given any deterministic effectively nonblocking consensus procedure P with two or more processes tolerating a single failure, we can **effectively construct** a non-terminating execution of it.

FLP as a Corollary

The proof is to use the Initialization Lemma to find a bivalent starting state b and then use the One Step Lemma to create an unbounded sequence of bivalent states.

Corollary (FLP): There is no single-failure responsive, deterministic consensus algorithm (terminating consensus procedure) on two or more processes.

Corollary (Strong FLP)*: Given any nonblocking deterministic consensus procedure on two or more processes, it has a non-terminating execution.

Conclusion

We initiate code generation at a very high level of abstraction by formally proving that **designs are realizable**.

By starting at such a high level, we discovered more correct options than possible by less technically advanced methods. **This discovery reveals new reasons for working formally at high levels of abstraction.**

Development of Event Logic

Our Event Logic is an abstract account of distributed computing inspired by the work of Winskel and Plotkin in the 80's on Petri Nets. It spans from the very abstract notion of **Event Class** down to formal models of protocols that can be compiled to **Message Automata** and from them into code in languages such as Java, ML, Erlang, F#, and so forth.

Safety

We almost have a **proof** that our design at the level of event classes meets the requirements.

We also need to know property P2, that two decided values agree even if no processes fail.

Suppose that at some P_i the $2f+1$ values collected in L are the same and likewise at P_j for $j \neq i$. Are the decided values equal?

Conclusions from Lectures

What I hope you have learned:

Propositions-as-types, Proofs-as-terms, Proofs-as-programs is a **collection of ideas** that relate logic, type theory, programming languages, and programming practice in a **theoretically deep and practical way**.

Conclusions continued

The **semantic approach** to type theory is a flexible way to justify new types and principles of reasoning about them.

Abstraction is the key to understanding complexity and rich type systems provide it.

There are many **fundamental theoretical questions** about type theory and its role in computer science and mathematics.

Conclusions

As Conor McBride says and illustrates, the Propositions-as-types ideas welcome you to **freely mix proofs and types into programming** and to use whatever lens is appropriate to the task at hand.

Foundational Criteria

What is required for a type theory to be an adequate **foundation for computer science**?

1. Proofs-as-programs works and the theory is a **programming language** and **programming logic** combined that can be well implemented.
2. **Computational mathematics**, e.g. numerical methods, computational geometry and algebra etc. is grounded in this theory.

Foundational Criteria continued

3. The theory can provide a **semantics to any programming language**.
4. All axioms and inference rules have a computational meaning, justified by the **propositions-as-types** principle and method.
5. The theory explains and justifies the **principles of computing** as they unfold.
6. Reasoning can well **automated well**.
7. The theory can be read classically as well.

Research Topics and Questions

Here are some interesting research questions and topics related to this lecture course.

1. Apply the **per semantics** to CIC, justify co-induction and classical semantics.
2. Formalize and implement Stuart Allen's approach to per semantics, possibly in CTT with a new axiom, say Consistent(CTT).
3. Rework the CTT framework , e.g. use **big step semantics** or start with a computation system based on **heaps**, integrate ideas from separation logic.

Research Topics and Questions continued

4. Explain the connection between **logical relations** and per semantics.
5. Give an **internal definition of recursive types** as Bickford and I did for co-recursive types, perhaps using constructive ordinals (the W type).
6. Put Coq **type checking** algorithm into CTT terms by adding optional annotations to the terms.

Research Topics and Questions continued

7. Look for **new types** that help organize and systematize the large Type Zoo of CTT. The type of **dependent intersection** should be an inspiration to us all. Kopylov was motivated by trying to improve on Hickey's **very dependent function type**. Try to raise the level of abstraction.
8. Read our article on the **Semantics of Reflected Proof** and use the ideas to prove key tactics correct and thus save “proof time”.

Research Topics and Questions continued

9. As you are learning from other lecturers in this summer school, these types theories can be made into real programming languages. Agda and Coq are examples. Use some of these ideas to **make an implicitly typed extensional theory like CTT into a good programming language.**
10. Specialize topic 9 to **distributed computing** based on our new process model and event logic.

THE END

Partial Functions

The concept of a **partial function** is an example of how challenging it is to include all computation in the object theory. It is also key to including unsolvability results with a minimum effort; the **halting problem** and related concepts are fundamentally about whether computations converge, and in type theory this is the essence of partiality. For example, **we do not know that the $3x+1$ function belongs to the type $\mathbb{N} \rightarrow \mathbb{N}$.**

Partial Functions

We do however know that the $3x+1$ function, call it f in this slide, is a **partial function** from numbers to numbers, thus for any n , $f(n)$ is a number if it converges (halts).

In CTT we say that a value a belongs to the **bar type** \bar{A} provided that it belongs to A if it converges. So f belongs to $A \rightarrow \bar{A}$ for $\bar{A} = \mathbb{N}$.

Unsolvable Problems

It is remarkable that we can prove that there is no function in CTT that can solve the convergence problem for elements of basic bar types.

We will show this for non empty type \bar{A} with element \bar{a} that converges in A for basic types such as Z , N , $\text{list}(A)$, etc. We rely on the typing that if F maps \bar{A} to \bar{A} , then $\text{fix}(F)$ is in \bar{A} .

Unsolvable Problems

Suppose there is a function h that decides halting. Define the following element of \bar{A} :

$$d = \text{fix}(\lambda(x. \text{if } h(x) \text{ then } \uparrow \text{ else } \bar{a} \text{ fi}))$$

where \uparrow is a diverging element, say $\text{fix}(\lambda(x.x))$.

Now we ask for the value of $h(d)$ and find a contradiction as follows:

Generalized Halting Problem

Suppose that $h(d) = t$, then d converges, but according to its definition, the result is the diverging computation \uparrow because by computing the fix term for one step, we reduce

$$d = \text{fix}(\lambda(x. \text{if } h(x) \text{ then } \uparrow \text{ else } \bar{a} \text{ fi}))$$

to $d = \text{if } h(d) \text{ then } \uparrow \text{ else } \bar{a} \text{ fi} .$

If $h(d) = f$, then we see that d converges to \bar{a} .

Why is this result noteworthy?

First notice that the result applies to any purported halting function h . In classical mathematics, there surely is a **noncomputable** function to decide halting.

Moreover the standard way to present unsolvability constructively is to model Turing machines and prove that **no Turing computable function can solve the halting problem**. But this result says that **no function can solve it**.

Definition

Attack-tolerant distributed systems change their protocols on-the-fly in response to apparent attacks from the environment; they substitute **functionally equivalent components** possibly more resistant to detected threats.

Definition

A **system** is built from **components** which consist of **processes** (protocols, algorithms).

system

component

process

Definition

A system is **correct-by-construction** if we create a correctness proof for it while creating the code. This happens if we **synthesize the program** from a constructive proof that the specification is realizable.

Main Result

We have found ways to automatically produce **many provably equivalent variants of components** using formal synthesis.

Variation arises from different choices made during the **proof and code synthesis process** starting from formal specifications.

A Discovery

In the course of this work we also discovered that it is possible to create **undefeatable attackers** for deterministic fault-tolerant consensus protocols.

Code diversity can protect against these attackers as well.

Key Lemma

One Step Lemma: Given any bivalent global state b of an effectively nonblocking consensus procedure P , and any process P_i , we can find an extension b' of b which is bivalent via Q_i .