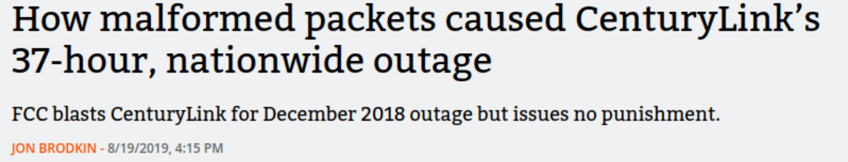
Formally Verified Compilation of Probabilistic Programs

Joseph Tassarotti (Boston College ⇒ New York University)

Joint work with:
Jean-Baptiste Tristan (Boston College)
Sam Stites (Northeastern University)

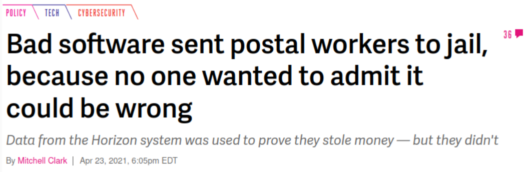
Bugs in software are costly and common



How malformed packets caused CenturyLink's 37-hour, nationwide outage

FCC blasts CenturyLink for December 2018 outage but issues no punishment.

JON BRODKIN - 8/19/2019, 4:15 PM

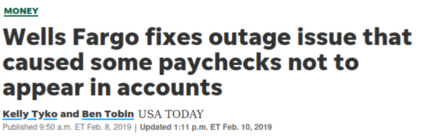


POLICY TECH CYBERSECURITY

Bad software sent postal workers to jail, because no one wanted to admit it could be wrong

Data from the Horizon system was used to prove they stole money — but they didn't

By Mitchell Clark | Apr 23, 2021, 6:05pm EDT



MONEY

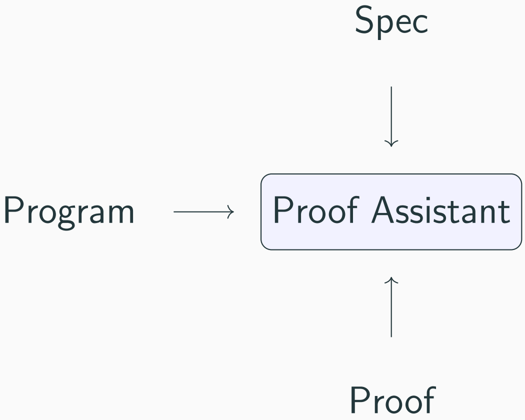
Wells Fargo fixes outage issue that caused some paychecks not to appear in accounts

Kelly Tyko and Ben Tobin USA TODAY

Published 9:50 a.m. ET Feb. 8, 2019 | **Updated 1:11 p.m. ET Feb. 10, 2019**

Formal verification

Alternative: **verify** software with **machine-checked proofs**.



Spec

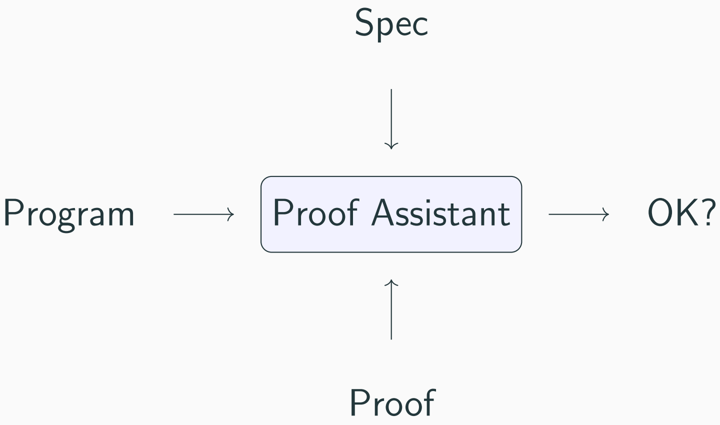
Program

Proof Assistant

Proof

Formal verification

Alternative: **verify** software with **machine-checked proofs**.



Spec

Program

Proof Assistant

OK?

Proof

Program Verification with a Proof Assistant

Program

```
func g() {
  x = f();
  if x > 0 {
    ...
  }
  else {
    ...
  }
}
```

Proof Assistant

```
Theorem gcorrect : ...
Proof.
  apply fcorrect.
  case (x > 0).
  + ...
    ...
  + ...
    ...
Qed.
```

Program Verification with a Proof Assistant

Program

```
func g() {
  x = f();
  if x > 0 {
    ...
  }
  else {
    ...
  }
}
```

Proof Assistant

```
Theorem gcorrect : ...
Proof.
  apply fcorrect.
  case (x > 0).
  + ...
    ...
  + ...
    ...
Qed.
```

Verification of Compilers

This methodology has been applied to build **verified** compilers:

- **CompCert** – C compiler, verified in Coq
- **CakeML** – ML compiler, verified in HOL

Proofs show that **semantics** of programs is preserved by compilation

ProbCompCert Project

Goal: build a verified compiler for Stan probabilistic programming language

- Modular design to make verification feasible
- Connects to CompCert for end-to-end guarantees

ProbCompCert Project

Goal: build a verified compiler for Stan probabilistic programming language

- Modular design to make verification feasible
- Connects to CompCert for end-to-end guarantees

Challenges:

- Randomized behavior
- Complex runtime
- More advanced mathematical foundations

Specification and Semantics

(Conventional) Compiler Correctness

A compiler C is a function that translates between languages:

$$C : \text{High Level Language} \rightarrow \text{Low Level Language}$$

(Conventional) Compiler Correctness

A compiler C is a function that translates between languages:

$$C : \text{High Level Language} \rightarrow \text{Low Level Language}$$

Compiled programs should **refine** original:

$$C(e) \sqsubseteq e$$

$\approx$ " Behaviors of $C(e)$ should be a subset of allowed behaviors of e "

What is the “behavior” of a Stan program?

A Stan program can be “run” in several different ways:

- MCMC Sampling
- Automatic Differentiation Variational Inference
- Maximum Likelihood Estimation

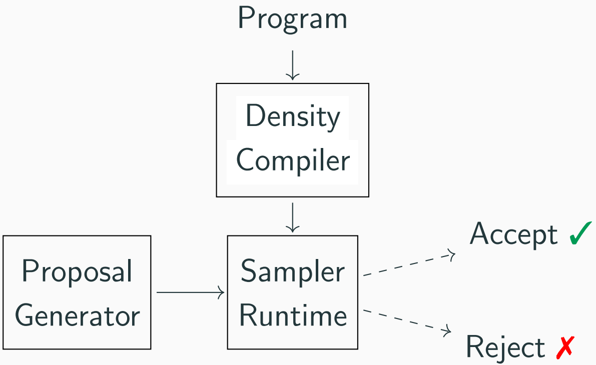
What is the “behavior” of a Stan program?

A Stan program can be “run” in several different ways:

- MCMC Sampling
- Automatic Differentiation Variational Inference
- Maximum Likelihood Estimation

Current focus: MCMC

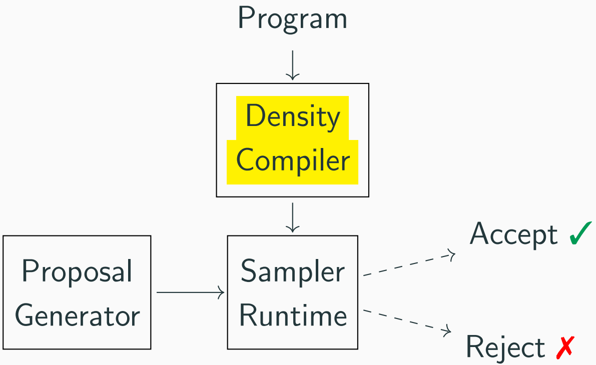
Samples generated with Markov Chain Monte Carlo (MCMC):



Asymptotically, distribution of samples approximates posterior

Current focus: MCMC

Samples generated with Markov Chain Monte Carlo (MCMC):



Asymptotically, distribution of samples approximates posterior

Model Block and Density Compiler

Stan programs are divided into **blocks**. The core is the **model block**

```
model {
  alpha ~ normal(0.0, 1.0);
  beta ~ normal(0.0, 1.0);
  y ~ normal(alpha + beta * x, 1.0);
}
```

Specifies how to compute **log posterior density**:

$$\text{model} : \text{data} \times \text{parameters} \to \mathbb{R}$$

Density compiler generates executable code for this function.

Models are Density Functions

Important: Stan does not restrict to “generative process” modeling.

Special variable called `target` is modified by sampling:

```
alpha ~ normal(0.0, 1.0)
```

is equivalent to

```
target += normal_lpdf(alpha | 0.0, 1.0);
```

`target` is implicitly returned at end of `model`

Defining the Semantics

ProbCompCert uses a hybrid semantics, two step process:

1. **Operational** – small-step rules for computation of expressions/statements, à la CompCert
2. **Denotational** – define probability distribution based on model block

(Heavily inspired by Gorinova et al. POPL 2019)

Defining the Semantics

ProbCompCert uses a hybrid semantics, two step process:

1. **Operational** – small-step rules for computation of expressions/statements, à la CompCert
2. **Denotational** – define probability distribution based on model block

(Heavily inspired by Gorinova et al. POPL 2019)

Integrating the Density

Model block specifies a (log) probability density (up to a constant).

Given data y, obtain a distribution on parameters by integrating + normalizing:

$$P(A) \propto \int_A e^{\text{model}(y,p)} d\mu(p)$$

Formalizing Semantics in Coq

Do not want to formalize lots of measure theory in Coq.

Stan requires parameters to be continuous. So improper Riemann integral over **rectangular** subsets suffices:

$$\int_A e^{\mathrm{model}(y,p)} d\mu(p) = \int_{a_1}^{b_1} \cdots \int_{a_n}^{b_n} e^{\mathrm{model}(y,x_1,\ldots,x_n)} dx_1 \cdots dx_n$$

Semantic Preservation

Let $e = (\text{model}, \text{data}, \text{parameters})$ be a Stan program. The **denotation** $[\![e]\!]$ is this function from data to Measure(parameters).

Goal: density compiler should **preserve** denotation:

$$[\![C(e)]\!] = [\![e]\!]$$

Compiler Verification Strategy

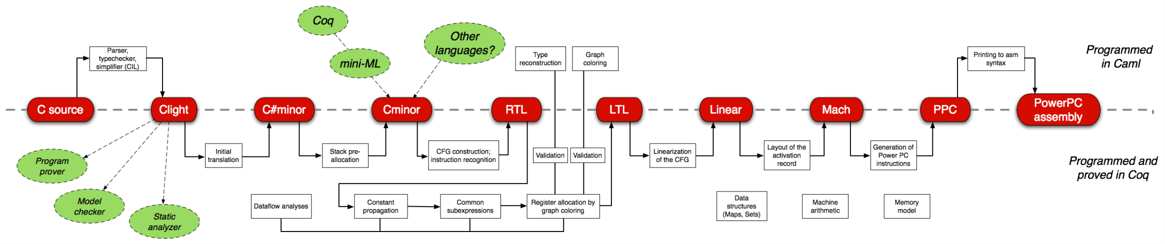
How to verify the compiler?

Adopt two important ideas from CompCert:

- Break compilation into many small passes.
- Forward simulation as a proof technique.

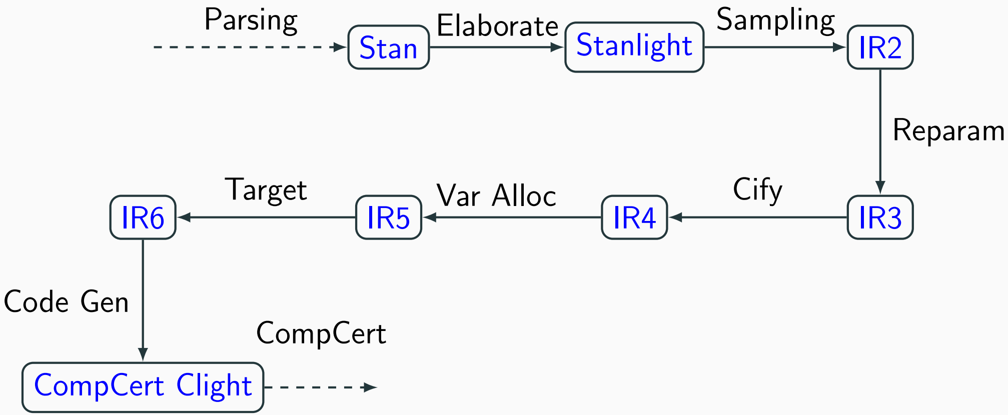
CompCert’s Pipeline

Many small passes between intermediate languages



Advantage: modular proofs of each pass

ProbCompCert's Pipeline



Simulation

Canonical proof technique is **backward** simulation:

$$\begin{tikzcd} \text{Source:} & e_1 \arrow[d, dotted, no head, "R"'] \\ \text{Compiled:} & e_2 \end{tikzcd}$$

Simulation

Canonical proof technique is **backward** simulation:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[d, dash, dotted, "R"'] & \\
\text{Compiled:} & e_2 \arrow[r] & e_2'
\end{tikzcd}$$

Simulation

Canonical proof technique is **backward** simulation:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r, "*"] \arrow[d, dotted, no head, "R"'] & e_1' \arrow[d, dotted, no head, "R"'] \\
\text{Compiled:} & e_2 \arrow[r] & e_2'
\end{tikzcd}$$

Simulation

Canonical proof technique is **backward** simulation:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r, "*"] \arrow[d, dotted, no head, "R"'] & e_1' \arrow[d, dotted, no head, "R"'] & \\
\text{Compiled:} & e_2 \arrow[r] & e_2' \arrow[r] & e_2''
\end{tikzcd}$$

Simulation

Canonical proof technique is **backward** simulation:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r, "*"] \arrow[d, dotted, no head, "R"'] & e_1' \arrow[r, "*"] \arrow[d, dotted, no head, "R"'] & e_1'' \arrow[d, dotted, no head, "R"'] & \\
\text{Compiled:} & e_2 \arrow[r] & e_2' \arrow[r] & e_2'' \arrow[r] & \cdots
\end{tikzcd}$$

Simulation

Canonical proof technique is **backward** simulation:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r, "*"] \arrow[d, dash, dotted, "R"'] & e_1' \arrow[r, "*"] \arrow[d, dash, dotted, "R"'] & e_1'' \arrow[r, "*"] \arrow[d, dash, dotted, "R"'] & \cdots \arrow[r] & v_1 \arrow[d, dash, dotted, "R"'] \\
\text{Compiled:} & e_2 \arrow[r] & e_2' \arrow[r] & e_2'' \arrow[r] & \cdots \arrow[r] & v_2
\end{tikzcd}$$

Forward Simulation

If program is **deterministic** can go other direction:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[d, dash, dotted, "R"'] \\
\text{Compiled:} & e_2
\end{tikzcd}$$

Forward Simulation

If program is **deterministic** can go other direction:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r] \arrow[d, dash, dotted, "R"'] & e_1' \\
\text{Compiled:} & e_2 &
\end{tikzcd}$$

Forward Simulation

If program is **deterministic** can go other direction:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r] \arrow[d, dotted, no head, "R"'] & e_1' \arrow[d, dotted, no head, "R"'] \\
\text{Compiled:} & e_2 \arrow[r, "*"] & e_2'
\end{tikzcd}$$

Forward Simulation

If program is **deterministic** can go other direction:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r] \arrow[d, dotted, no head, "R"'] & e_1' \arrow[r] \arrow[d, dotted, no head, "R"'] & e_1'' \\
\text{Compiled:} & e_2 \arrow[r, "*"] & e_2' &
\end{tikzcd}$$

Forward Simulation

If program is **deterministic** can go other direction:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r] \arrow[d, dotted, no head, "R"'] & e_1' \arrow[r] \arrow[d, dotted, no head, "R"'] & e_1'' \arrow[d, dotted, no head, "R"'] \\
\text{Compiled:} & e_2 \arrow[r, "*"] & e_2' \arrow[r, "*"] & e_2''
\end{tikzcd}$$

Forward Simulation

If program is **deterministic** can go other direction:

$$\begin{tikzcd}
\text{Source:} & e_1 \arrow[r] \arrow[d, dotted, no head, "R"'] & e_1' \arrow[r] \arrow[d, dotted, no head, "R"'] & e_1'' \arrow[r] \arrow[d, dotted, no head, "R"'] & \cdots \arrow[r] & v_1 \arrow[d, dotted, no head, "R"'] \\
\text{Compiled:} & e_2 \arrow[r, "*"] & e_2' \arrow[r, "*"] & e_2'' \arrow[r, "*"] & \cdots \arrow[r] & v_2
\end{tikzcd}$$

Forward Simulation for ProbCompCert?

How can we use forward simulation if MCMC sampler is randomized?

Forward Simulation for ProbCompCert?

How can we use forward simulation if MCMC sampler is randomized?

Answer: runtime is randomized, BUT model block code is **deterministic**

Forward Simulation Preserves Denotation

Recall: we want $[\![C(e)]\!] = [\![e]\!]$. That is, $\forall y, A$:

$$\int_A e^{C(\text{model})(y,p)} d\mu(p) \propto \int_A e^{\text{model}(y,p)} d\mu(p)$$

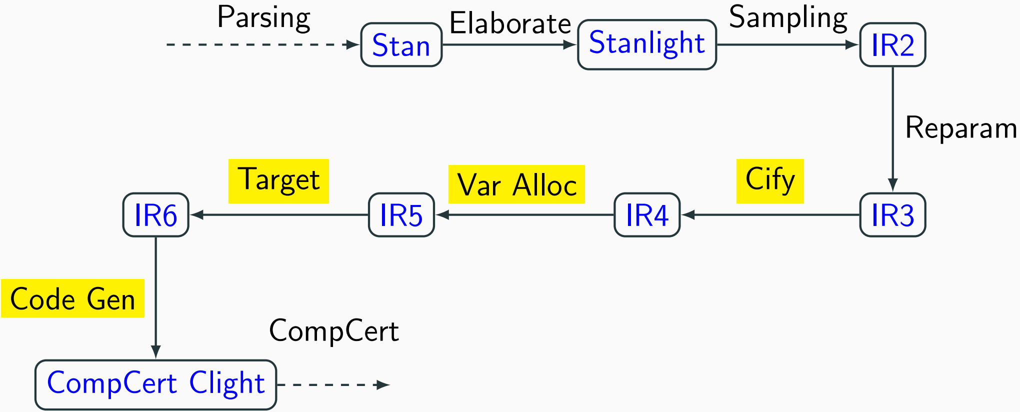
Forward Simulation Preserves Denotation

Recall: we want $[\![C(e)]\!] = [\![e]\!]$. That is, $\forall y, A$:

$$\int_A e^{C(\text{model})(y,p)} d\mu(p) \propto \int_A e^{\text{model}(y,p)} d\mu(p)$$

Standard simulation implies $C(\text{model})(y, p) = \text{model}(y, p)$ so integral preserved.

Passes Covered by Standard Simulation



1. Sample Statement Pass

Sample Statement Pass desugars:

```
alpha ~ normal(x, y)
```

into

```
target += normal_lpdf(alpha | x, y);
```

1. Sample Statement Pass

Sample Statement Pass desugars:

```
alpha ~ normal(x, y)
```

into

```
target += normal_lpdf(alpha | x, y);
```

BUT Stan also drops addition of constants to `target`:

```
target += f(...) + D   ⇒   target += f(...)
```

Why is dropping additive constants sound?

We get the same distribution after normalizing:

$$\begin{aligned}\int_A e^{\text{model}(y,p)+D}\,d\mu(p) &= \int_A e^{D}\cdot e^{\text{model}(y,p)}\,d\mu(p)\\ &\propto \int_A e^{\text{model}(y,p)}\,d\mu(p)\end{aligned}$$

2. Reparameterization

Stan allows parameters to have a **constrained** range:

```
real<lower=0> alpha;
real<upper=1> beta;
real<lower=0, upper=1> gamma;
```

However, sampler operates over **unconstrained** parameter space

2. Reparameterization

To bridge gap, compiler inserts code to map between constrained and unconstrained.

E.g. to handle `real<lower=b>alpha`:

1. Sample **unconstrained** `alpha’`
2. Insert prologue in model to set `alpha = exp(alpha') + b`

2. Reparameterization

To bridge gap, compiler inserts code to map between constrained and unconstrained.

E.g. to handle `real<lower=b>alpha`:

1. Sample **unconstrained** `alpha’`
2. Insert prologue in model to set `alpha = exp(alpha') + b`
3. Add Jacobian offset: `target += alpha'`

2. Reparameterization

To bridge gap, compiler inserts code to map between constrained and unconstrained.

E.g. to handle `real<lower=b>alpha`:

1. Sample **unconstrained** `alpha’`
2. Insert prologue in model to set `alpha = exp(alpha') + b`
3. Add Jacobian offset: `target += alpha'`

Where does this come from? Integral change of variables:

$$\int_a^b f(\varphi(x))\varphi'(x)dx = \int_{\varphi(a)}^{\varphi(b)} f(x)dx$$

Using Simulations

These passes change **extensional** behavior of `model`, but can still use simulation:

1. **Operational Proof:** simulation to show that

$$C(\mathit{model})(y, p) = g(\mathit{model}(y, p))$$

Using Simulations

These passes change **extensional** behavior of `model`, but can still use simulation:

1. **Operational Proof:** simulation to show that

$$C(model)(y,p) = g(model(y,p))$$

2. **Denotational Proof:** show that

$$\int_A e^{g(\text{model}(y,p))} d\mu(p) \propto \int_A e^{\text{model}(y,p)} d\mu(p)$$

Challenges and Conclusion

Open Question

Denotational arguments assume **exact** real arithmetic.

Reality: approximate **floating point** arithmetic.

$$\int_A e^{\mathrm{model}(y,p)} d\mu(p) \Rightarrow \sum_{p \in A} e^{\mathrm{model}(y,p)} \mu(p)$$

Open Question

Denotational arguments assume **exact** real arithmetic.

Reality: approximate **floating point** arithmetic.

$$\int_A e^{\mathrm{model}(y,p)} d\mu(p) \Rightarrow \sum_{p \in A} e^{\mathrm{model}(y,p)} \mu(p)$$

How to address?

- Don't?
- Bound floating point error?
- Implement an exact arithmetic backend?

Conclusion

Goal: build a verified compiler for Stan probabilistic programming language

- Modular design to make verification feasible
- Connects to CompCert for end-to-end guarantees

Phase 1: Density Compilation

- Many intermediate passes
- Leverage forward simulations as much as possible