Dynamic Logic for Practical Program Verification

Set 5

Wolfgang Ahrendt

Chalmers University of Technology

VTSA Summer School, Luxembourg, 2024

Verification of Solidity

- Ongoing work: Guilherme Alvares, Wolfgang Ahrendt, Richard Bubel
- First results:
  Ahrendt, Bubel
  Functional Verification of Smart Contracts via Strong Data Integrity
  published in ISoLA'20

Part I

Dynamic Logic and Calculus for Solidity

Deductive verification of genuine Solidity

- Specification/verification methodology for Solidity smart contracts

Deductive verification of genuine Solidity

- Specification/verification methodology for Solidity smart contracts
- Invariant based

Deductive verification of genuine Solidity

- Specification/verification methodology for Solidity smart contracts
- Invariant based
- Specify/verify strong data integrity properties

Deductive verification of genuine Solidity

- Specification/verification methodology for Solidity smart contracts
- Invariant based
- Specify/verify strong data integrity properties
- Reason about arbitrary usage of contracts (incl. re-entrance)

Deductive verification of genuine Solidity

- Specification/verification methodology for Solidity smart contracts
- Invariant based
- Specify/verify strong data integrity properties
- Reason about arbitrary usage of contracts (incl. re-entrance)
- Functional verification wrt. payment history

Deductive verification of genuine Solidity

- Specification/verification methodology for Solidity smart contracts
- Invariant based
- Specify/verify strong data integrity properties
- Reason about arbitrary usage of contracts (incl. re-entrance)
- Functional verification wrt. payment history
- KeY for Solidity:

Deductive verification of genuine Solidity

- Specification/verification methodology for Solidity smart contracts
- Invariant based
- Specify/verify strong data integrity properties
- Reason about arbitrary usage of contracts (incl. re-entrance)
- Functional verification wrt. payment history
- KeY for Solidity: SolidiKeY

Deductive verification of genuine Solidity

- Specification/verification methodology for Solidity smart contracts
- Invariant based
- Specify/verify strong data integrity properties
- Reason about arbitrary usage of contracts (incl. re-entrance)
- Functional verification wrt. payment history
- KeY for Solidity: SolidiKeY (prototype)

Motivating Example: Auction Contract

```
contract Auction {

 address payable private auctionOwner;
 mapping (address => uint) public bid;
 mapping (address => bool) private bidded;

 ...

 function placeOrIncreaseBid() public payable {
  // Call example: auct.placeOrIncreaseBid{value:5}()

  bid[msg.sender] = bid[msg.sender] + msg.value;

  // If the caller didn't bid yet, add her to bidders
  if (!bidded[msg.sender]) {
    bidders[numberOfBidders] = msg.sender;
    ...
  }
 }
 ...
```

Auction Contract (cont'd)

```
...

function withdraw() public {
  // A bidder can withdraw all her money

  require(bidded[msg.sender]);

  msg.sender.transfer(bid[msg.sender]);
  bid[msg.sender] = 0;
}

...
```

Auction Contract (cont'd)

```
  ...

  function closeAuction() public {
    // Determine highestBid and winner
    ...
    // Transfer the money to the auction owner
    auctionOwner.transfer(highestBid);
    bid[winner] = 0;

    // Reimburse everyone else
    for(i = 0; i < numberOfBidders; i = i + 1) {
      bidders[i].transfer(bid[bidders[i]]);
      bid[bidders[i]] = 0;
    }
  }
}
```

What does it mean for `Auction` to be correct?

Implementation maintains, at critical points, a contract invariant:

$$\forall a \in \text{address}.$$

What does it mean for `Auction` to be correct?

Implementation maintains, at critical points, a contract invariant:

$$\forall a \in \text{address}.$$

$$\text{bid[a]}$$

$$=$$

What does it mean for `Auction` to be correct?

Implementation maintains, at critical points, a contract invariant:

$$\forall a \in \text{address}.$$

$$\text{bid[a]}$$

$$=$$

$$\sum\{\text{msg.value} \mid \text{msg.sender} = \text{a}\}$$

What does it mean for `Auction` to be correct?

Implementation maintains, at critical points, a contract invariant:

$$\forall a \in \text{address}.$$

$$\text{bid[a]}$$

$$=$$

$$\sum\{\text{msg.value} \mid \text{msg.sender} = \text{a}\}$$

$$-$$

$$\sum\{v \mid \text{a.transfer}(v)\}$$

What does it mean for `Auction` to be correct?

Implementation maintains, at critical points, a contract invariant:

$$\forall a \in \text{address}.$$

$$\text{bid[a]}$$

$$=$$

$$\sum\{\text{msg.value} \mid \text{msg.sender} = \text{a}\}$$

$$-$$

$$\sum\{v \mid \text{a.transfer}(v)\}$$

Contract invariant defines relation between:

- SC's local state
- communication history

What does it mean for `Auction` to be correct?

Implementation maintains, at critical points, a contract invariant:

$$\forall a \in \text{address}.$$

$$\text{bid}[a]$$

$$=$$

$$\sum\{\text{msg.value} \mid \text{msg.sender} = a\}$$

$$-$$

$$\sum\{v \mid a.\text{transfer}(v)\}$$

Contract invariant defines relation between:

- SC's local state
- communication history = payment history

What does it mean for `Auction` to be correct?

Implementation maintains, at critical points, a contract invariant:

$$\forall a \in \text{address}.$$

$$\text{bid[a]}$$

$$=$$

$$\sum\{\text{msg.value} \mid \text{msg.sender} = \text{a}\}$$

$$-$$

$$\sum\{v \mid \text{a.transfer}(v)\}$$

Contract invariant defines relation between:

- SC's local state
- communication history = payment history

Data integrity $\equiv$ contract invariant holds at critical points

Critical Points?

Which are the critical points of a contract's code?

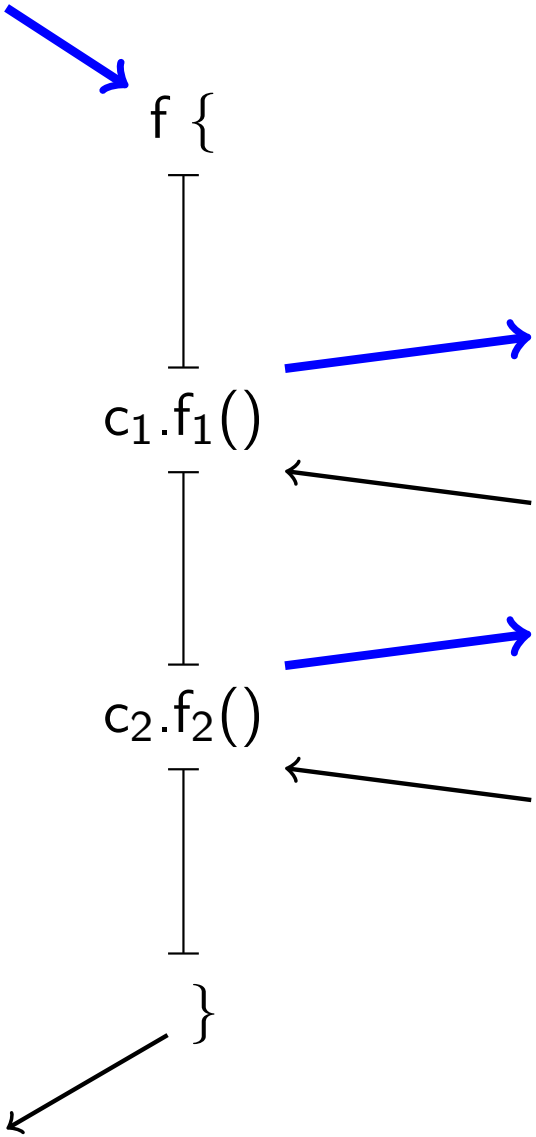
Critical Points?

Which are the critical points of a contract's code?

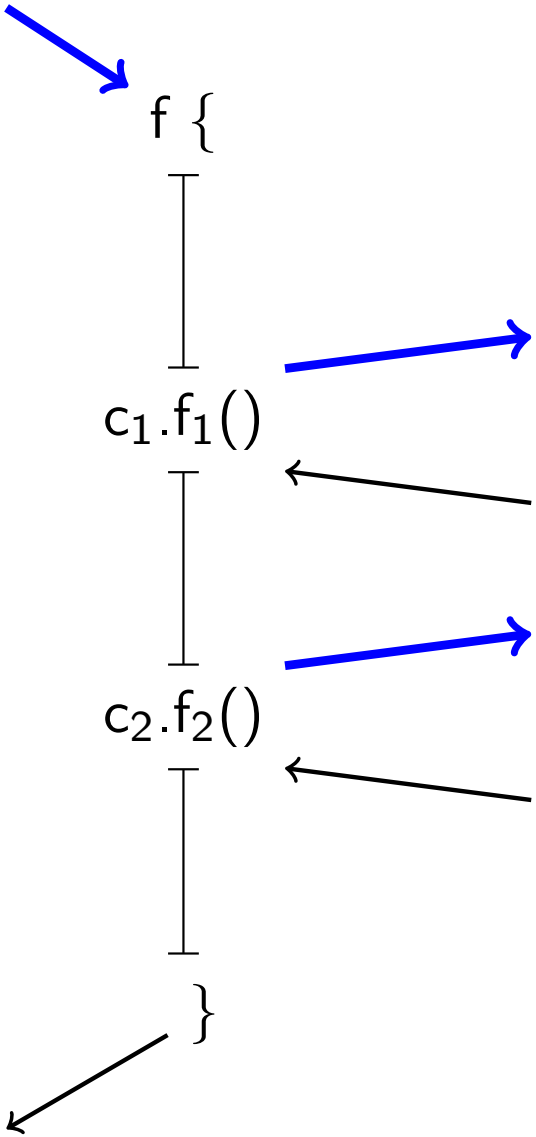
Revisit Auction:

```
contract Auction {

  ...

  function withdraw() public {
    // A bidder can withdraw all her money

    require(bidded[msg.sender]);

    msg.sender.transfer(bid[msg.sender]);
    bid[msg.sender] = 0;
  }

  ...
}
```

Proof Methodology

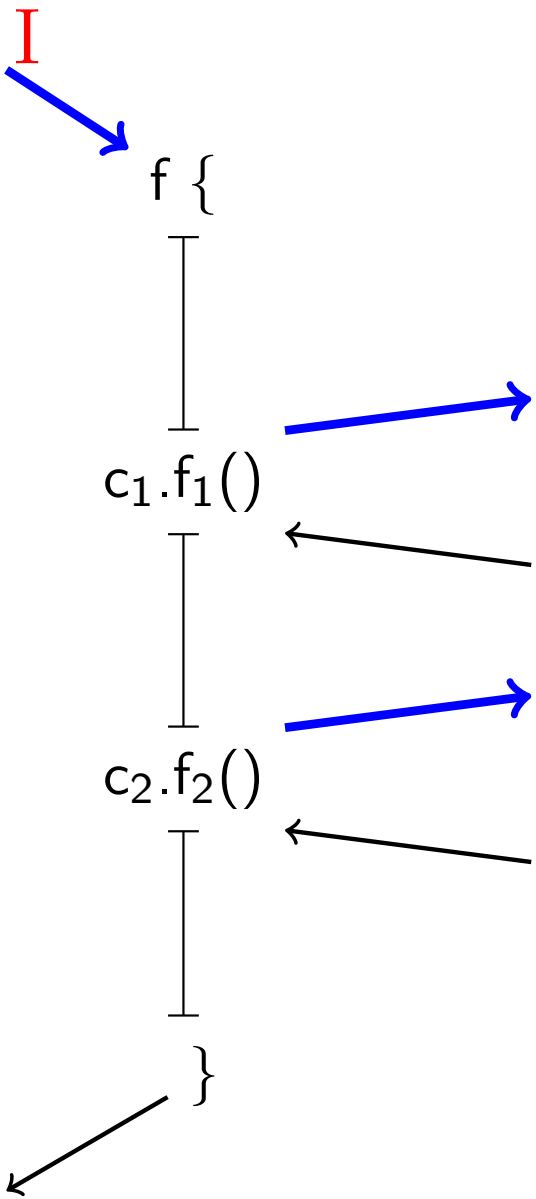


Proof Methodology



Invariant must hold whenever control is *outside* contract:

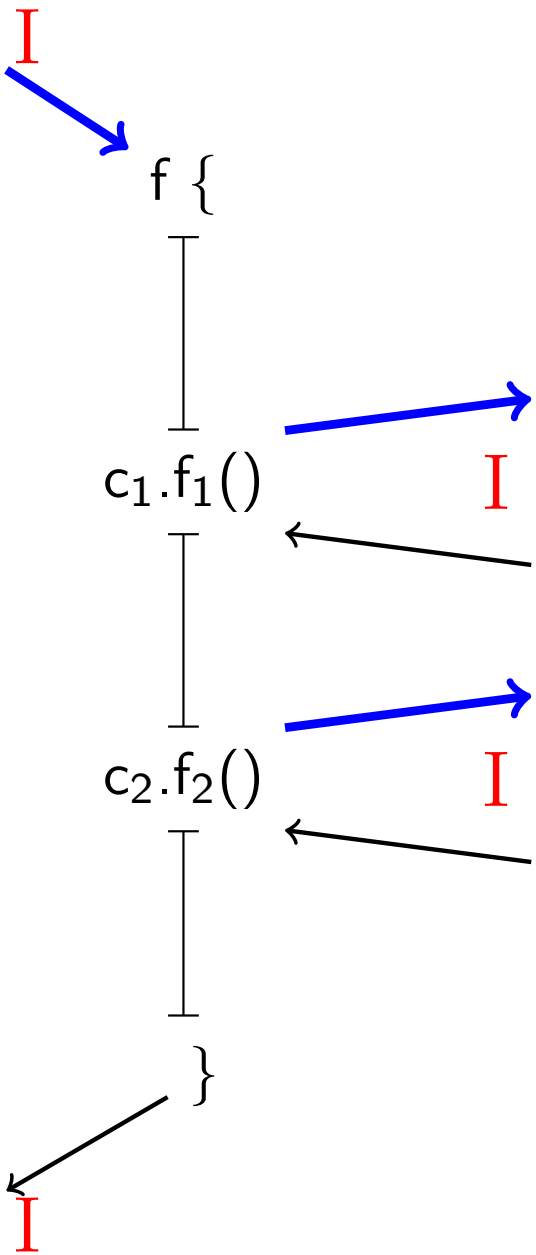
Proof Methodology



Invariant must hold whenever control is *outside* contract:

- Invariant is assumed when gaining control, but **before** money is received

Proof Methodology



Invariant must hold whenever control is *outside* contract:

- Invariant is assumed when gaining control, but **before** money is received
- Invariant has to be shown when ceasing control, but **after** money is sent

Capturing the History of Money-Flow

Verification with explicit communication histories complex

Capturing the History of Money-Flow

Verification with explicit communication histories complex

Instead:
Reasoning about a mapping

$$\text{net} : \text{address} \to \mathbb{Z}$$

$$\text{net}(a) = \begin{cases} \text{money received by } \textit{this} \text{ contract from } a \\ - \\ \text{money sent by } \textit{this} \text{ contract to } a \end{cases}$$

Contract Invariant of Auction

$$\exists hb \in \text{address}.\ \forall a \in \text{address}.$$
$$\text{bid}[hb] \geq \text{bid}[a]$$

Contract Invariant of Auction

$$
\begin{aligned}
&\exists hb \in \mathsf{address}.\ \forall a \in \mathsf{address}.\\
&\qquad \mathsf{bid}[hb] \geq \mathsf{bid}[a]\\
&\wedge \quad (\ a \neq hb \wedge a \neq \mathsf{auctionOwner} \rightarrow\\
&\qquad\ \ \mathsf{bid}[a] = \mathsf{net}(a)\ )\\
&\wedge\\
&\vdots\\
&\vdots
\end{aligned}
$$

Contract Invariant of Auction

$$
\begin{aligned}
&\exists hb \in \text{address}.\ \forall a \in \text{address}.\\
&\quad\ \text{bid}[hb] \geq \text{bid}[a]\\
&\wedge\ \ (\ a \neq hb \wedge a \neq \text{auctionOwner} \rightarrow\\
&\qquad \text{bid}[a] = \text{net}(a)\ )\\
&\wedge\\
&\vdots\\
&\vdots
\end{aligned}
$$

Advantage of using net mapping

$$\mathsf{bid}[a] = \mathsf{net}(a)$$

replaces the following equation

Advantage of using net mapping

$$\mathsf{bid}[a] = \mathsf{net}(a)$$

replaces the following equation

$$\mathsf{bid}[\mathsf{a}]$$

$$=$$

$$\sum\{\mathsf{msg.value} \mid \mathsf{msg.sender} = \mathsf{a}\}$$

$$-$$

$$\sum\{v \mid \mathsf{a.transfer}(v)\}$$

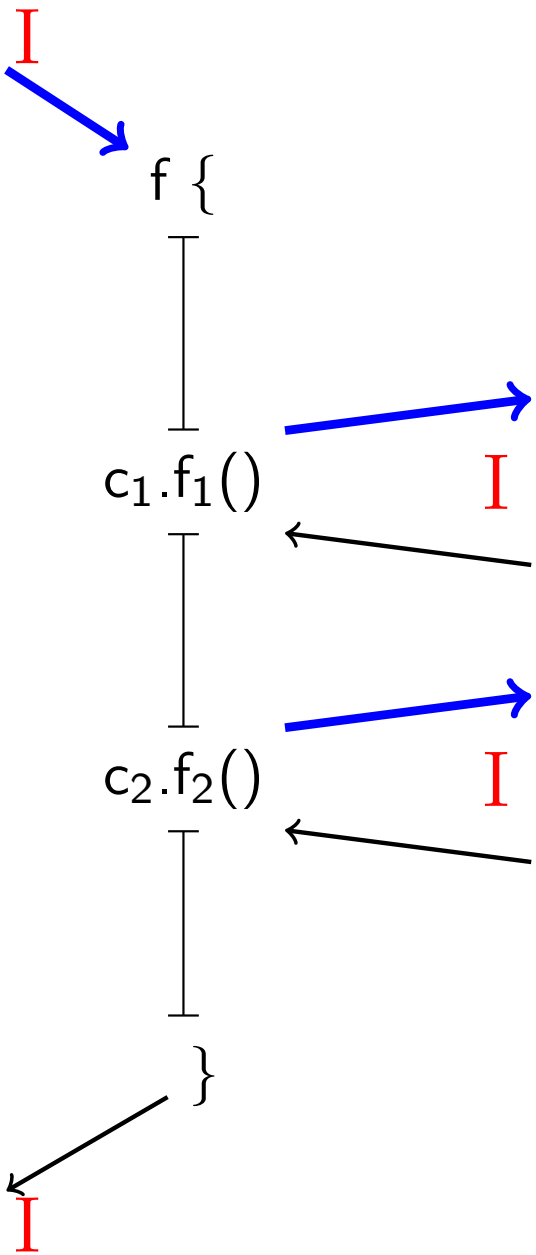
Contract Invariant of Auction

$$
\begin{aligned}
&\exists hb \in \text{address}.\ \forall a \in \text{address}.\\
&\quad\ \text{bid}[hb] \geq \text{bid}[a]\\
&\wedge\ \ (\ a \neq hb \wedge a \neq \text{auctionOwner} \rightarrow\\
&\qquad \text{bid}[a] = \text{net}(a)\ )\\
&\wedge\\
&\vdots\\
&\vdots
\end{aligned}
$$

Contract Invariant of Auction

$$
\begin{aligned}
&\exists hb \in \text{address}.\ \forall a \in \text{address}.\\
&\quad \text{bid}[hb] \geq \text{bid}[a]\\
&\wedge \quad (\ a \neq hb \wedge a \neq \text{auctionOwner} \rightarrow\\
&\qquad \text{bid}[a] = \text{net}(a)\ )\\
&\wedge \quad \text{bid}[hb] = \text{net}(hb) + \text{net}(\text{auctionOwner})\\
&\vdots\\
&\vdots
\end{aligned}
$$

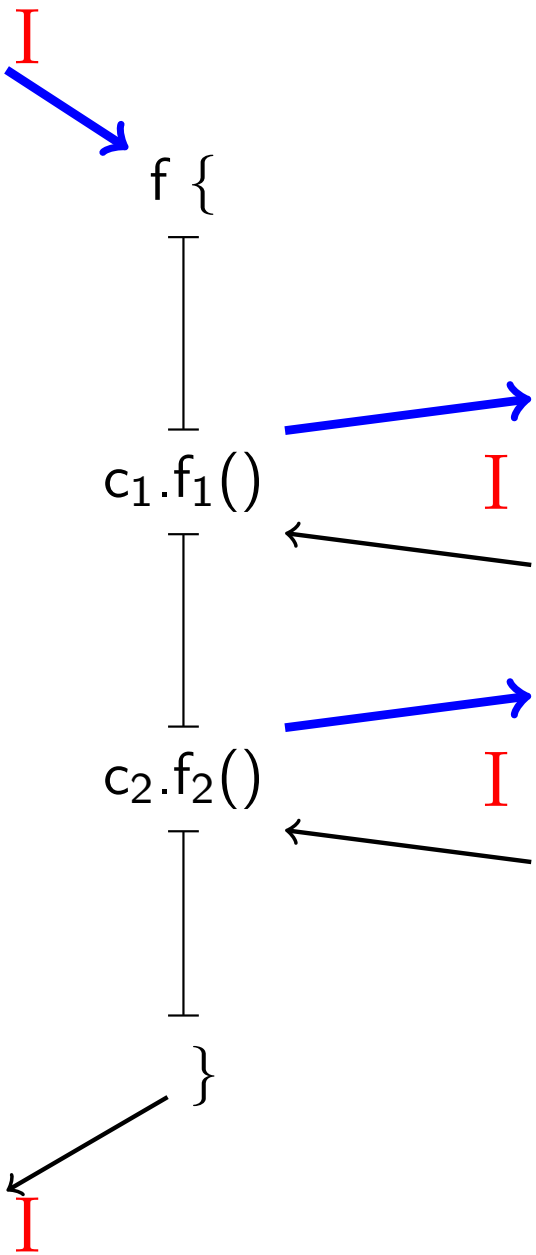
Proof Methodology



Invariant must hold whenever control is *outside* contract:

- Invariant is assumed when gaining control, but **before** money is received
- Invariant has to be shown when ceasing control, but **after** money is sent

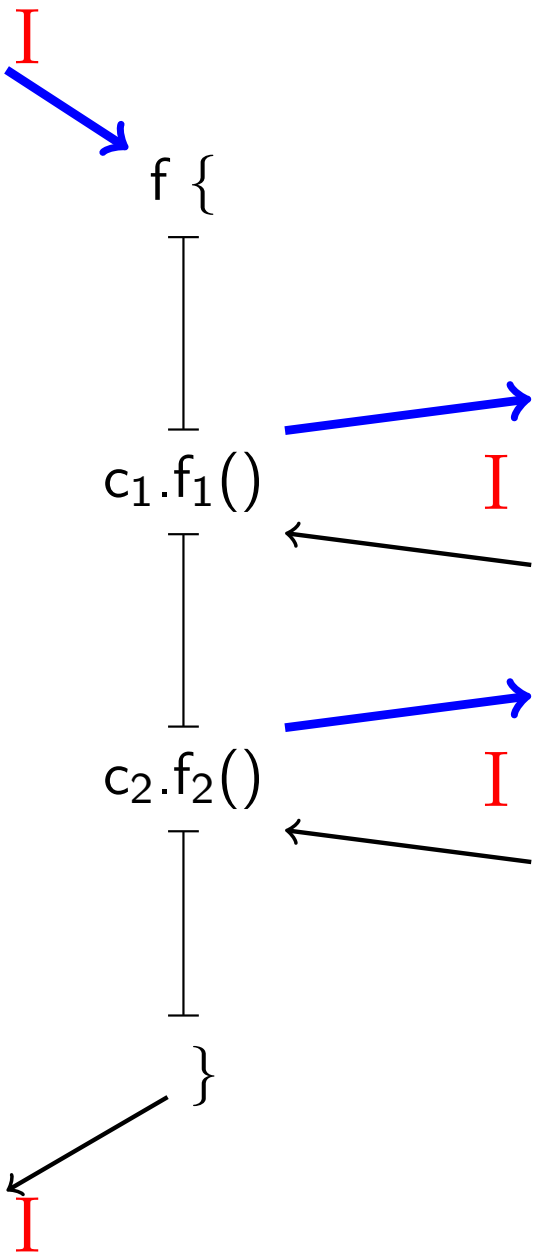
Proof Methodology



Invariant must hold whenever control is *outside* contract:

- Invariant is assumed when gaining control, but **before** money is received
- Invariant has to be shown when ceasing control, but **after** money is sent
- In I, we use net instead of history

Proof Methodology



Invariant must hold whenever control is *outside* contract:

- Invariant is assumed when gaining control, but **before** money is received
- Invariant has to be shown when ceasing control, but **after** money is sent
- In I, we use net instead of history
- net is modified at function start and at calls

Part II

Solidity Dynamic Logic

Solidity Dynamic Logic

Solidity DL extends first-order logic with two operators:

- $[\mathsf{p}]\phi$
- $\langle \mathsf{p} \rangle \phi$

where p is Solidity code, ϕ another Solidity DL formula.

Solidity Dynamic Logic

Solidity DL extends first-order logic with two operators:

- $[\mathrm{p}]\phi$
- $\langle \mathrm{p} \rangle \phi$

where p is Solidity code, ϕ another Solidity DL formula.
(Often, ϕ is a first-order formula.)

Solidity Dynamic Logic

Solidity DL extends first-order logic with two operators:

- $[\mathrm{p}]\phi$
- $\langle \mathrm{p} \rangle \phi$

where p is Solidity code, ϕ another Solidity DL formula.
(Often, ϕ is a first-order formula.)

Meaning:

Solidity Dynamic Logic

Solidity DL extends first-order logic with two operators:

- $[\mathrm{p}]\phi$
- $\langle \mathrm{p} \rangle \phi$

where p is Solidity code, ϕ another Solidity DL formula.
(Often, ϕ is a first-order formula.)

Meaning:

- $[\mathrm{p}]\phi$: If p executes *successfully* then ϕ holds afterwards (partial correctness)

Solidity Dynamic Logic

Solidity DL extends first-order logic with two operators:

- $[\mathrm{p}]\phi$
- $\langle \mathrm{p} \rangle \phi$

where p is Solidity code, ϕ another Solidity DL formula.
(Often, ϕ is a first-order formula.)

Meaning:

- $[\mathrm{p}]\phi$: If p executes *successfully* then ϕ holds afterwards (partial correctness)
- $\langle \mathrm{p} \rangle \phi$: p executes *successfully* and ϕ holds afterwards (total correctness)

Solidity Dynamic Logic

Solidity DL extends first-order logic with two operators:

- $[p]\phi$
- $\langle p \rangle \phi$

where p is Solidity code, ϕ another Solidity DL formula.
(Often, ϕ is a first-order formula.)

Meaning:

- $[p]\phi$: If p executes *successfully* then ϕ holds afterwards (partial correctness)
- $\langle p \rangle \phi$: p executes *successfully* and ϕ holds afterwards (total correctness)

Successful execution: no exception thrown, no state reverted.

Solidity Dynamic Logic

Solidity DL extends first-order logic with two operators:

- $[p]\phi$
- $\langle p \rangle \phi$

where p is Solidity code, ϕ another Solidity DL formula.
(Often, ϕ is a first-order formula.)

Meaning:

- $[p]\phi$: If p executes *successfully* then ϕ holds afterwards (partial correctness)
- $\langle p \rangle \phi$: p executes *successfully* and ϕ holds afterwards (total correctness)

Successful execution: no exception thrown, no state reverted.

What about non-termination?

Calculus Rules: require

Rules for `require`

$$\frac{}{\Gamma \Longrightarrow [\texttt{require}(\mathsf{b});\ \omega]\phi, \Delta}$$

Calculus Rules: require

Rules for `require`

$$\frac{\Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{require}(\mathsf{b});\ \omega]\phi, \Delta}$$

Calculus Rules: require

Rules for `require`

$$\frac{\Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{require}(\mathsf{b});\ \omega]\phi, \Delta}$$

assume $\mathsf{b} = \mathit{true}$ when verifying remaining code

Calculus Rules: require

Rules for `require`

$$\frac{\Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{require}(\mathsf{b});\ \omega]\phi, \Delta} \quad \mathsf{b} \text{ simple}$$

assume $\mathsf{b} = \mathit{true}$ when verifying remaining code

Calculus Rules: require

Rules for `require`

$$\frac{\Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{require(b);}\ \omega]\phi, \Delta}\ \text{b simple}$$

assume $\mathsf{b} = \mathit{true}$ when verifying remaining code

$$\frac{\Gamma \Longrightarrow [\texttt{bool b; b = e; require(b);}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{require(e);}\ \omega]\phi, \Delta}\ \text{e complex}$$

Calculus Rules: require

Rules for `require`

$$\frac{\Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{require}(\mathsf{b});\ \omega]\phi, \Delta}\ \text{b simple}$$

assume b = *true* when verifying remaining code

$$\frac{\Gamma \Longrightarrow [\texttt{bool b; b = e; require}(\mathsf{b});\ \omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{require}(\mathsf{e});\ \omega]\phi, \Delta}\ \text{e complex}$$

expression e symbolically executed first

Calculus Rules: assert

Rules for `assert`

$$\frac{}{\Gamma \Longrightarrow [\texttt{assert}(\mathsf{b});\ \omega]\phi, \Delta}$$

Calculus Rules: assert

Rules for `assert`

$$\frac{\Gamma \Longrightarrow \mathsf{b} = \mathit{true}, \Delta}{\Gamma \Longrightarrow [\texttt{assert}(\mathsf{b});\ \omega]\phi, \Delta}$$

Calculus Rules: assert

Rules for `assert`

$$\frac{\Gamma \Longrightarrow \mathsf{b} = \mathit{true}, \Delta \qquad \Gamma, \qquad \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{assert}(\mathsf{b});\ \omega]\phi, \Delta}$$

Calculus Rules: assert

Rules for `assert`

$$\frac{\Gamma \Longrightarrow \mathsf{b} = \mathit{true}, \Delta \qquad \Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{assert}(\mathsf{b});\ \omega]\phi, \Delta}$$

Calculus Rules: assert

Rules for `assert`

$$\frac{\Gamma \Longrightarrow \mathsf{b} = \mathit{true}, \Delta \qquad \Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{assert}(\mathsf{b});\ \omega]\phi, \Delta}$$

prove $\mathsf{b} = \mathit{true}$ assume $\mathsf{b} = \mathit{true}$ when verifying remaining code

Calculus Rules: assert

Rules for `assert`

$$\frac{\Gamma \Longrightarrow \mathsf{b} = \mathit{true}, \Delta \qquad \Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{assert}(\mathsf{b});\ \omega]\phi, \Delta}\ \mathsf{b}\ \text{simple}$$

prove $\mathsf{b} = \mathit{true}$ assume $\mathsf{b} = \mathit{true}$ when verifying remaining code

Calculus Rules: assert

Rules for `assert`

$$\frac{\Gamma \Longrightarrow \mathsf{b} = \mathit{true}, \Delta \qquad \Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{assert}(\mathsf{b});\ \omega]\phi, \Delta}$$ b simple

prove $\mathsf{b} = \mathit{true}$ assume $\mathsf{b} = \mathit{true}$ when verifying remaining code

$$\frac{\Gamma \Longrightarrow [\texttt{bool b; b = e; assert(b);}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{assert(e);}\ \omega]\phi, \Delta}$$ e complex

Calculus Rules: assert

Rules for `assert`

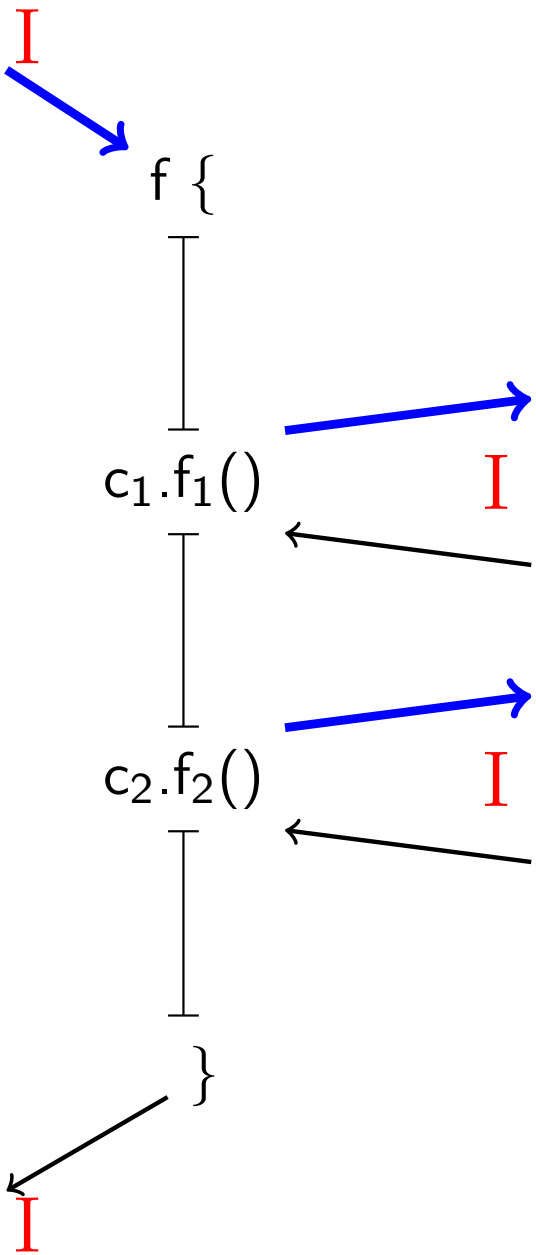
$$\frac{\Gamma \Longrightarrow \mathsf{b} = \mathit{true}, \Delta \qquad \Gamma, \mathsf{b} = \mathit{true} \Longrightarrow [\omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{assert}(\mathsf{b});\ \omega]\phi, \Delta}$$ b simple

prove $\mathsf{b} = \mathit{true}$ assume $\mathsf{b} = \mathit{true}$ when verifying remaining code

$$\frac{\Gamma \Longrightarrow [\texttt{bool b; b = e; assert(b);}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow [\texttt{assert(e);}\ \omega]\phi, \Delta}$$ e complex

expression e symbolically executed first

Recall: Proof Methodology



Invariant must hold whenever control is *outside* contract:

- Invariant is assumed when gaining control, but **before** money is received
- Invariant has to be shown when ceasing control, but **after** money is sent
- In I, we use net instead of history
- net is modified at function start and at calls

Calculus Rules: transfer

Rule for `transfer`

$$\frac{}{\Gamma \Longrightarrow [\mathsf{a}.\mathtt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Calculus Rules: transfer

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}\, \mathrm{I}, \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\mathtt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Prove I *after* money has been sent. (Why?)

Calculus Rules: transfer

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}\, \mathrm{I}, \Delta \qquad \Gamma \Longrightarrow \{\mathsf{anon}(\mathsf{storage})\}(\mathrm{I} \to [\omega]\phi), \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\texttt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Prove I *after* money has been sent. (Why?)

Once **`transfer`** returns, forget the entire state, assume I, and verify $[\omega]\phi$.

Verification Attempt of Auction Contract shows:

```
contract Auction {

  ...

  function withdraw() public {
    // A bidder can withdraw all her money

    require(bidded[msg.sender]);

    msg.sender.transfer(bid[msg.sender]);
    bid[msg.sender] = 0;
  }

  ...
}
```

Verification Attempt of Auction Contract shows:

```
contract Auction {

  ...

  function withdraw() public {
    // A bidder can withdraw all her money

    require(bidded[msg.sender]);

    msg.sender.transfer(bid[msg.sender]);
    bid[msg.sender] = 0;
  }

  ...
}
```

Invariant broken here.
Re-entrance attack possible.

Fixed Auction Contract

```
contract Auction {

  ...

  function withdraw() public {
    // A bidder can withdraw all her money

    require(bidded[msg.sender]);

    uint tmp = bid[msg.sender];
    bid[msg.sender] = 0;
    msg.sender.transfer(tmp);
  }

  ...
}
```

Calculus Rules: transfer

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}\, \mathrm{I}, \Delta \qquad \Gamma \Longrightarrow \{\mathsf{anon}(\mathsf{storage})\}(\mathrm{I} \rightarrow [\omega]\phi), \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\texttt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Prove I *after* money has been sent. (Why?)

Once **`transfer`** returns, forget the entire state, assume I, and verify $[\omega]\phi$.

Calculus Rules: transfer

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}\,\mathrm{I}, \Delta \qquad \Gamma \Longrightarrow \{\mathsf{anon}(\mathsf{storage})\}(\mathrm{I} \to [\omega]\phi), \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\texttt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Prove I *after* money has been sent. (Why?)
Once `transfer` returns, forget the entire state, assume I, and verify $[\omega]\phi$.

This rule applies if execution of `transfer` can potentially call back.

Calculus Rules: transfer

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}\, \mathrm{I}, \Delta \qquad \Gamma \Longrightarrow \{\mathsf{anon}(\mathsf{storage})\}(\mathrm{I} \to [\omega]\phi), \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\texttt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Prove I *after* money has been sent. (Why?)
Once `transfer` returns, forget the entire state, assume I, and verify $[\omega]\phi$.

This rule applies if execution of `transfer` can potentially call back.

Currently, `transfer` *cannot* call back. (2300 gas, too little for calls)

Calculus Rules: transfer

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}\,\mathrm{I}, \Delta \qquad \Gamma \Longrightarrow \{\mathsf{anon}(\mathsf{storage})\}(\mathrm{I} \to [\omega]\phi), \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\texttt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Prove I *after* money has been sent. (Why?)
Once `transfer` returns, forget the entire state, assume I, and verify $[\omega]\phi$.

This rule applies if execution of `transfer` can potentially call back.

Currently, `transfer` *cannot* call back. (2300 gas, too little for calls)

But, according to some blogs, "gas costs can and will change".

Calculus Rules: transfer (call-back possible)

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}\, \mathrm{I}, \Delta \qquad \Gamma \Longrightarrow \{\mathsf{anon}(\mathsf{storage})\}(\mathrm{I} \to [\omega]\phi), \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\texttt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Prove I *after* money has been sent. (Why?)
Once `transfer` returns, forget the entire state, assume I, and verify $[\omega]\phi$.

This rule applies if execution of `transfer` can potentially call back.

Currently, `transfer` *cannot* call back. (2300 gas, too little for calls)

But, according to some blogs, "gas costs can and will change".

Calculus Rules: transfer (no call-back possible)

Rule for `transfer`

$$\frac{}{\Gamma \Longrightarrow [\mathsf{a}.\texttt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

This rule applies to *current* Ethereum.
Execution of `transfer` cannot call back.

Calculus Rules: transfer (no call-back possible)

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\mathtt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Send the money, and verify $[\omega]\phi$ afterwards.

This rule applies to *current* Ethereum.
Execution of `transfer` cannot call back.

Calculus Rules: transfer (no call-back possible)

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\mathtt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Send the money, and verify $[\omega]\phi$ afterwards.
No need to show I. (Why?)

This rule applies to *current* Ethereum.
Execution of `transfer` cannot call back.

Calculus Rules: transfer (no call-back possible)

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\mathtt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Send the money, and verify $[\omega]\phi$ afterwards.
No need to show I. (Why?)
No need to assume I. (Why?)

This rule applies to *current* Ethereum.
Execution of `transfer` cannot call back.

Calculus Rules: transfer (no call-back possible)

Rule for `transfer`

$$\frac{\Gamma \Longrightarrow \{\mathsf{net}(\mathsf{a}) := \mathsf{net}(\mathsf{a}) - \mathsf{v}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow [\mathsf{a}.\texttt{transfer}(\mathsf{v});\ \omega]\phi, \Delta}$$

Send the money, and verify $[\omega]\phi$ afterwards.
No need to show I. (Why?)
No need to assume I. (Why?)

This rule applies to *current* Ethereum.
Execution of `transfer` cannot call back.

SolidiKeY supports both options (as taclet options)

Part III

Concluding Remarks

- DL offers a framework for practical verification

- DL offers a framework for practical verification
- DL well suited for source code level and system level verification

- DL offers a framework for practical verification
- DL well suited for source code level and system level verification
- Many relevant properties *naturally* expressed in DL

- DL offers a framework for practical verification
- DL well suited for source code level and system level verification
- Many relevant properties *naturally* expressed in DL
- DL combines well with symbolic execution

- DL offers a framework for practical verification
- DL well suited for source code level and system level verification
- Many relevant properties *naturally* expressed in DL
- DL combines well with symbolic execution
- DL combines well with test generation

- DL offers a framework for practical verification
- DL well suited for source code level and system level verification
- Many relevant properties *naturally* expressed in DL
- DL combines well with symbolic execution
- DL combines well with test generation
- DL combines well with runtime verification

- DL offers a framework for practical verification
- DL well suited for source code level and system level verification
- Many relevant properties *naturally* expressed in DL
- DL combines well with symbolic execution
- DL combines well with test generation
- DL combines well with runtime verification
- Competitive tooling of DL

Thanks!