Nomos: Resource-Aware Session Types for Programming Digital Contracts

Stephanie Balzer, Ankush Das, Jan Hoffmann, and Frank Pfenning

Programming language developed at Carnegie Mellon

With some slides from Ankush.
Digital Contracts (or Smart Contracts)

**Smart contracts (Ethereum): programs stored on a blockchain**

- Carry out (financial) transactions between (untrusted) agents
- Cannot be modified but have state
  - Community needs to reach consensus on the result of execution
  - *Users need to pay for the execution cost upfront*
Digital Contracts (or Smart Contracts)

Smart contracts (Ethereum): programs stored on a blockchain

- Carry out (financial) transactions between (untrusted) agents
- Cannot be modified but have state
  - Community needs to reach consensus on the result of execution
  - Users need to pay for the execution cost upfront

![Diagram showing the process of executing a smart contract]

- User initiates a transaction with money (estimated gas cost)
- Transaction is sent to the miner
- Miner processes the transaction
- Remaining gas is determined
- miner checks if sufficient gas is available
- New block is created if sufficient gas is available

smart contract new block
Bugs in Digital Contracts are Expensive

- Bugs result in financial disasters (DAO, Parity Wallet, King of Ether, …)
- Bugs are difficult to fix because they alter the contract
Can Programming Languages Prevent Bugs?
Can Programming Languages Prevent Bugs?

Yes!
Can Programming Languages Prevent Bugs?

Example: memory safety

• Most security vulnerabilities are based on memory safety issues (Microsoft: 70% over past 12 years in MS products)

• Why stick with unsafe languages? Legacy code, developers (training, social factors, …)
Can Programming Languages Prevent Bugs?

Example: memory safety

• Most security vulnerabilities are based on memory safety issues (Microsoft: 70% over past 12 years in MS products)

• Why stick with unsafe languages? Legacy code, developers (training, social factors, …)

Languages for Digital Contracts

• Great opportunity to start from a clean slate

• Correctness and readability of contracts are priorities
Can Programming Languages Prevent Bugs?

**Example: memory safety**

- Most security vulnerabilities are based on memory safety issues (Microsoft: 70% over past in the past 12 years in MS products)
- Why stick with unsafe languages? Legacy code, developers (training, social factors, …)

**Languages for Digital Contracts**

- Great opportunity to start from a clean slate
- Correctness and readability of contracts are priorities

**Nomos**

- Build on state-of-the art: statically-typed, strict, functional language
- Address domain-specific issues
Domain Specific Bugs: Auction Contract

status: running
Domain Specific Bugs: Auction Contract

Bidder 1

Bid 1

Bidder 2

Bid 2

Bidder 3

Bid 3

status: running
Domain Specific Bugs: Auction Contract

Bidder 1

Bidder 2

Bidder 3

Bid 1
Bid 2
Bid 3

status: running
Domain Specific Bugs: Auction Contract

Bidder 1

Bidder 2

Bidder 3

Bid 1

Bid 2

Bid 3

status: ended
Domain Specific Bugs: Auction Contract

Bidder 1

Bidder 2

Bidder 3

Bid 1

Bid 2

Bid 3

status: ended
Domain Specific Bugs: Auction Contract

Bidder 1
Bid 1

Bidder 2
Bid 2

Bidder 3
Bid 3

status: ended
Auction Contract in Solidity

```solidity
function bid() public payable {
    bid = msg.value;
    bidder = msg.sender;
    pendingReturns[bidder] = bid;
    if (bid > highestBid) {
        highestBidder = bidder;
        highestBid = bid;
    }
}

function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```

What happens if collect is called when auction is running?
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```

What happens if `collect` is called when auction is running?

Add require (status == ended);
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```

What happens if `collect` is called when auction is running?

Protocol is not statically enforced!

add require (status == ended);
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```

What happens if collect is called twice?
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```

What happens if `collect` is called twice?

```
set pendingReturns[msg.sender] = 0
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    return true;
}
```

What happens if `collect` is called twice?

Linearity is not enforced!

```solidity
set pendingReturns[msg.sender] = 0
```
Auction in Solidity

```solidity
class Auction {
    function collect() public returns (bool) {
        require (msg.sender != highestBidder);
        require (status == ended);
        uint amount = pendingReturns[msg.sender];
        msg.sender.send(amount);
        pendingReturns[msg.sender] = 0;
        return true;
    }
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    pendingReturns[msg.sender] = 0;
    return true;
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    pendingReturns[msg.sender] = 0;
    return true;
}
```

Method ‘send’ potentially transfers control to other contract.
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    pendingReturns[msg.sender] = 0;
    return true;
}
```

Method ‘send’ potentially transfers control to other contract.

‘send’ should be the last instruction.
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    msg.sender.send(amount);
    pendingReturns[msg.sender] = 0;
    return true;
}
```

Method ‘send’ potentially transfers control to other contract.

Re-entrancy attack

‘send’ should be the last instruction.
**Auction in Solidity**

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    pendingReturns[msg.sender] = 0;
    msg.sender.send(amount);
    return true;
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    pendingReturns[msg.sender] = 0;
    msg.sender.send(amount);
    return true;
}
```
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    pendingReturns[msg.sender] = 0;
    msg.sender.send(amount);
    return true;
}
```

Method ‘send’ potentially transfers control to other contract.
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    pendingReturns[msg.sender] = 0;
    msg.sender.send(amount);
    return true;
}
```

Method ‘send’ potentially transfers control to other contract.

Need to check return value of ‘send’.
Auction in Solidity

```solidity
function collect() public returns (bool) {
    require (msg.sender != highestBidder);
    require (status == ended);
    uint amount = pendingReturns[msg.sender];
    pendingReturns[msg.sender] = 0;
    msg.sender.send(amount);
    return true;
}
```

Method ‘send’ potentially transfers control to other contract.

Out-of-gas exception.

Need to check return value of ‘send’.
Domain-Specific Issues with Digital Contracts

1. Resource consumption (gas cost)
   - Participants have to agree on the result of a computation
     ➡ Denial of service attacks
     ➡ Would like to have static gas bounds
Domain-Specific Issues with Digital Contracts

1. *Resource consumption (gas cost)*
   - Participants have to agree on the result of a computation
     ➡ Denial of service attacks
     ➡ Would like to have static gas bounds

2. *Contract protocols and interfaces*
   - Contract protocols should be described and enforced
     ➡ Prevent issues like reentrancy bugs (DAO)
Domain-Specific Issues with Digital Contracts

1. **Resource consumption (gas cost)**
   - Participants have to agree on the result of a computation
     ➡ Denial of service attacks
     ➡ Would like to have static gas bounds

2. **Contract protocols and interfaces**
   - Contract protocols should be described and enforced
     ➡ Prevent issues like reentrancy bugs (DAO)

3. **Keeping track of assets (crypto coins)**
   - Assets should not be duplicated
   - Assets should not be lost
Nomos: A Type-Based Approach

A *statically-typed, strict, functional language*

- Functional fragment of ML

*Additional features for domain-specific requirements*

<table>
<thead>
<tr>
<th>Language feature</th>
<th>Expertise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gas bounds</td>
<td>Automatic amortized resource analysis</td>
</tr>
<tr>
<td>Tracking assets</td>
<td>Linear type system</td>
</tr>
<tr>
<td>Contract interfaces</td>
<td>Shared binary session types</td>
</tr>
</tbody>
</table>
Nomos: A Type-Based Approach

A statically-typed, strict, functional language

- Functional fragment of ML

Additional features for domain-specific requirements

<table>
<thead>
<tr>
<th>Language feature</th>
<th>Expertise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gas bounds</td>
<td>Automatic amortized resource analysis</td>
</tr>
<tr>
<td>Tracking assets</td>
<td>Linear type system</td>
</tr>
<tr>
<td>Contract interfaces</td>
<td>Shared binary session types</td>
</tr>
</tbody>
</table>

Based on a linear type system
1. Automatic amortized resource analysis (AARA)
Resource Bound Analysis

Given: A (functional) program P

Question: What is the (worst-case) resource consumption of P as a function of the size of its inputs?
Resource Bound Analysis

Given: A (functional) program P

Question: What is the (worst-case) resource consumption of P as a function of the size of its inputs?

Clock cycles, heap space, gas, ...
Resource Bound Analysis

Given: A (functional) program P

Question: What is the (worst-case) resource consumption of P as a function of the size of its inputs?

Clock cycles, heap space, gas, ...

Not only asymptotic bounds but concrete constant factors.
Automatic Resource Bound Analysis

Given: A (functional) program P

Question: What is the (worst-case) resource consumption of P as a function of the size of its inputs?

Clock cycles, heap space, gas, ...

Not only asymptotic bounds but concrete constant factors.
Resource Bound Analysis

Given: A (functional) program P

Question: What is the (worst-case) resource consumption of P as a function of the size of its inputs?

Clock cycles, heap space, gas, ...

Goal: produce proofs (easily checkable)

Not only asymptotic bounds but concrete constant factors.
AARA: Use Potential Method

- Assign potential functions to data structures
  - States are mapped to non-negative numbers

- Potential pays the resource consumption and the potential at the following program point

- Initial potential is an upper bound

\[ \Phi(state) \geq 0 \]

\[ \Phi(before) \geq \Phi(after) + \text{cost} \]

\[ \Phi(initial\ state) \geq \sum \text{cost} \]
AARA: Use Potential Method

- Assign potential functions to data structures
  - States are mapped to non-negative numbers
  \[ \Phi(state) \geq 0 \]

- Potential pays the resource consumption and the potential at the following program point
  \[ \Phi(before) \geq \Phi(after) + cost \]
  \[ \downarrow \text{telescoping} \]
  \[ \Phi(initial\ state) \geq \sum \text{cost} \]

Type systems for automatic analysis

- Fix a format of potential functions (basis like in linear algebra)

- Type rules introduce linear constraint on coefficients
AARA: Use Potential Method

- Assign potential functions to data structures
  - States are mapped to non-negative numbers
  \[ \Phi(\text{state}) \geq 0 \]

- Potential pays the resource consumption and the potential at the following program point
  \[ \Phi(\text{before}) \geq \Phi(\text{after}) + \text{cost} \]

- Initial potential is an upper bound
  \[ \Phi(\text{initial state}) \geq \sum \text{cost} \]

Type systems for automatic analysis

- Fix a format of potential functions (basis like in linear algebra)
- Type rules introduce linear constraint on coefficients

Example: Append for Persistent Lists

append(x, y)

Heap-space usage is 2n if

- n is the length of list x
- One list element requires two heap cells (data and pointer)
Example: Append for Persistent Lists

\[
\text{append}(x, y)
\]

Heap-space usage is \(2n\) if

- \(n\) is the length of list \(x\)
- One list element requires two heap cells (data and pointer)

Example evaluation:
Example: Append for Persistent Lists

Heap-space usage is 2n if

- n is the length of list x
- One list element requires two heap cells (data and pointer)

Example evaluation:

Example evaluation:
Example: Append for Persistent Lists

```
append(x, y)
```

Heap-space usage is \(2n\) if

- \(n\) is the length of list \(x\)
- One list element requires two heap cells (data and pointer)

Example evaluation:

```
x → a
    ↓
  b → c

y → d
    ↓
  c → e
```
Example: Append for Persistent Lists

Heap-space usage is \(2n\) if

- \(n\) is the length of list \(x\)
- One list element requires two heap cells (data and pointer)

Example evaluation:

\[
\text{append}(x, y)
\]
Example: Append for Persistent Lists

append(x, y)

Heap-space usage is \(2n\) if

- \(n\) is the length of list \(x\)
- One list element requires two heap cells (data and pointer)

Example evaluation:

\[
\begin{align*}
x & \rightarrow a \\
& \rightarrow b \\
& \rightarrow c \\
y & \rightarrow d \\
& \rightarrow c \\
& \rightarrow b \\
& \rightarrow a \\
& \rightarrow e
\end{align*}
\]
Example: Append for Persistent Lists

append(x, y)

Heap-space usage is 2n if
- n is the length of list x
- One list element requires two heap cells (data and pointer)

Example evaluation:

x -> a -> b -> c

y -> d -> c -> b -> a -> append(x, y)

e
Example: Append for Persistent Lists

```
append(x, y)
```

Heap-space usage is $2n$ if

- $n$ is the length of list $x$
- One list element requires two heap cells (data and pointer)

Example evaluation:

- $x$: $a \rightarrow b \rightarrow c$
- $y$: $d \rightarrow c \rightarrow b \rightarrow a \rightarrow \text{append}(x, y)$

Heap usage: $2n = 2 \times 3 = 6$
Example: Composing Calls of Append

\[
f(x,y,z) = \
\text{let } t = \text{append}(x,y) \text{ in} \
\text{append}(t,z)
\]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)
Example: Composing Calls of Append

\[ f(x,y,z) = \]
\[ \text{let } t = \text{append}(x,y) \text{ in append}(t,z) \]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if
\begin{itemize}
  \item n is the length of list x
  \item m is the length of list y
\end{itemize}

Initial potential: \( 4n + 2m = 4 \times 3 + 2 \times 2 = 16 \)
Example: Composing Calls of Append

\[
f(x, y, z) = \begin{aligned} 
\text{let } & t = \text{append}(x, y) \text{ in} \\
\text{append}(t, z) 
\end{aligned}
\]

Heap usage of \( f(x, y, z) \) is \( 2n + 2(n+m) \) if

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4n + 2m = 4 \times 3 + 2 \times 2 = 16 \)
Example: Composing Calls of Append

\[ f(x,y,z) = \]
\[
\text{let } t = \text{append}(x,y) \text{ in} \\
\text{append}(t,z) \\
\]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4n + 2m = 4 \times 3 + 2 \times 2 = 16 \)
Example: Composing Calls of Append

f(x,y,z) =
let t = append(x,y) in
append(t,z)

Heap usage of f(x,y,z) is 2n + 2(n+m) if
- n is the length of list x
- m is the length of list y

Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16
Example: Composing Calls of Append

\[ f(x,y,z) = \]
\[ \text{let } t = \text{append}(x,y) \text{ in } \text{append}(t,z) \]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4*n + 2*m = 4*3 + 2*2 = 16 \)
Example: Composing Calls of Append

\[ f(x,y,z) = \]
\[ \text{let } t = \text{append}(x,y) \text{ in append}(t,z) \]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4n + 2m = 4 \times 3 + 2 \times 2 = 16 \)
Example: Composing Calls of Append

\[ f(x, y, z) = \]
\[
\text{let } t = \text{append}(x, y) \text{ in} \]
\[
\text{append}(t, z) \]

Heap usage of \( f(x, y, z) \) is \( 2n + 2(n+m) \) if:

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4n + 2m = 4 \times 3 + 2 \times 2 = 16 \)
Example: Composing Calls of Append

\[
f(x,y,z) = \text{let } t = \text{append}(x,y) \text{ in } \text{append}(t,z)
\]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4\times n + 2\times m = 4\times 3 + 2\times 2 = 16 \)
Example: Composing Calls of Append

$f(x,y,z) =$
let $t = \text{append}(x,y)$ in
$\text{append}(t,z)$

Heap usage of $f(x,y,z)$ is $2n + 2(n+m)$ if

- $n$ is the length of list $x$
- $m$ is the length of list $y$

Initial potential: $4n + 2m = 4*3 + 2*2 = 16$
Example: Composing Calls of Append

\[ f(x,y,z) = \begin{array}{l}
\text{let } t = \text{append}(x,y) \text{ in } \\
\text{append}(t,z) \end{array} \]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4*n + 2*m = 4*3 + 2*2 = 16 \)
Example: Composing Calls of Append

\[
f(x,y,z) = \begin{aligned}
    \text{let } & t = \text{append}(x,y) \text{ in } \\
    \text{append}(t,z)
\end{aligned}
\]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if:

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4n + 2m = 4\times3 + 2\times2 = 16 \)
Example: Composing Calls of Append

\[ f(x, y, z) = \]
\[
\text{let } t = \text{append}(x, y) \text{ in } \text{append}(t, z)\
\]

Heap usage of \( f(x, y, z) \) is \( 2n + 2(n+m) \) if:
- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4n + 2m = 4 \times 3 + 2 \times 2 = 16 \)
Example: Composing Calls of Append

\[ f(x,y,z) = \]
\[ \text{let } t = \text{append}(x,y) \text{ in } \text{append}(t,z) \]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if:
- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4n + 2m = 4 \times 3 + 2 \times 2 = 16 \)
Example: Composing Calls of Append

\[
f(x,y,z) = 
\begin{align*}
& \text{let } t = \text{append}(x,y) \text{ in} \nonumber \\
& \text{append}(t,z)
\end{align*}
\]

Heap usage of \( f(x,y,z) \) is \( 2n + 2(n+m) \) if:

- \( n \) is the length of list \( x \)
- \( m \) is the length of list \( y \)

Initial potential: \( 4n + 2m = 4\times3 + 2\times2 = 16 \)

Implicit reasoning about size-changes.
Example: Composing Calls of Append

\[
f(x,y,z) = \begin{cases} 
\text{let } t = \text{append}(x,y) \text{ in } \\
\text{append}(t,z) 
\end{cases}
\]

The most general type of append is specialized at call-sites:

\[
\text{append: } (L^q \text{ (int)}, L^p \text{ (int)}) \rightarrow L^r \text{ (int)} \mid \phi
\]

Linear constraints.
## Polynomial Potential Functions

User-defined **resource metrics**
(i.e., by tick(q) in the code)

Naturally **compositional**: tracks size changes, types are specifications

Bound inference by reduction to efficient LP solving

**Type derivations prove bounds**
with respect to the cost semantics

<table>
<thead>
<tr>
<th>Linear Potential Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
</tr>
</tbody>
</table>

---

**Polynomial Potential Functions**
User-defined resource metrics (i.e., by tick(q) in the code)

Naturally compositional: tracks size changes, types are specifications

Bound inference by reduction to efficient LP solving

Type derivations prove bounds with respect to the cost semantics

Strong soundness theorem.

Polynomial Potential Functions
User-defined **resource metrics** (i.e., by tick(q) in the code)

Naturally **compositional**: tracks size changes, types are specifications

Bound inference by reduction to efficient **LP solving**

Type derivations prove bounds with respect to the cost semantics

---

**Strong soundness theorem.**
### Polynomial Potential Functions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Linear Potential Functions</th>
<th>Multivariate Polynomial Potential Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-defined <strong>resource metrics</strong> (i.e., by tick(q) in the code)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Naturally <strong>compositional</strong>: tracks size changes, types are specifications</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bound inference by reduction to efficient <strong>LP solving</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Type derivations prove bounds</strong> with respect to the cost semantics</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

For example $m \times n^2$.

**Strong soundness theorem.**

---

**Polynomial Potential Functions**
Implementations: RaML and Absynth

Resource Aware ML (RaML)
- Based on Inria’s OCaml compiler
- Polymorphic and higher-order functions
- User-defined data types
- Side effects (arrays and references)

Absynth
- Based on control-flow graph IR
- Different front ends
- Bounds are integer expressions
- Supports probabilistic programs

http://raml.co
<table>
<thead>
<tr>
<th>Algorithm Description</th>
<th>Computed Bound</th>
<th>Actual Behavior</th>
<th>Analysis Runtime</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting A-nodes (asort)</td>
<td>$11+22kn + 13k^2nv+13m + 15n$</td>
<td>O($k^2n+m$)</td>
<td>0.14 s</td>
<td>5656</td>
</tr>
<tr>
<td>Quick sort (lists of lists)</td>
<td>$3 -7.5nm + 7.5nm^2 + 19.5m + 16.5m^2$</td>
<td>O($nm^2$)</td>
<td>0.27 s</td>
<td>8712</td>
</tr>
<tr>
<td>Merge sort (list.ml)</td>
<td>$43 + 30.5n + 8.5n^2$</td>
<td>O($n \log n$)</td>
<td>0.11 s</td>
<td>3066</td>
</tr>
<tr>
<td>Split and sort</td>
<td>$11 + 47n + 29n^2$</td>
<td>O($n^2$)</td>
<td>0.69 s</td>
<td>3793</td>
</tr>
<tr>
<td>Longest common subsequence</td>
<td>$23 + 10n + 52nm + 25m$</td>
<td>O($nm$)</td>
<td>0.16 s</td>
<td>901</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>$3 + 2nm + 18m + 22mxy + 16my$</td>
<td>O($mxy$)</td>
<td>1.11 s</td>
<td>3901</td>
</tr>
<tr>
<td>Evaluator for boolean expressions (tutorial)</td>
<td>$10+11n+16m+16mx+16my+20x+20y$</td>
<td>O($mx+my$)</td>
<td>0.33 s</td>
<td>1864</td>
</tr>
<tr>
<td>Dijkstra’s shortest-path algorithm</td>
<td>$46 + 33n + 111n^2$</td>
<td>O($n^2$)</td>
<td>0.11 s</td>
<td>2808</td>
</tr>
<tr>
<td>Echelon form</td>
<td>$8 + 43m^2n + 59m + 63m^2$</td>
<td>O($nm^2$)</td>
<td>1.81 s</td>
<td>8838</td>
</tr>
<tr>
<td>Binary multiplication (CompCert)</td>
<td>$2+17kr+10ks+25k + 8l+2+7r+8$</td>
<td>O($kr+ks$)</td>
<td>14.04 s</td>
<td>89,507</td>
</tr>
<tr>
<td>Square root (CompCert)</td>
<td>$13+66m+16mn +4m^2 + 59n +4n^2$</td>
<td>O($n^2$)</td>
<td>18.25 s</td>
<td>135,529</td>
</tr>
</tbody>
</table>
Quick Sort for Integers

Evaluation-step bound vs.
measured behavior
Longest Common Subsequence

Evaluation-step bound vs. measured behavior
First automatically derived bound for LCS.
Automatic Amortized Resource Analysis (AARA)

Type system for deriving symbolic resource bounds

- **Compositional:** Integrated with type systems or program logics
- **Expressive:** Bounds are multivariate resource polynomials
- **Reliable:** Formal soundness proof wrt. cost semantics
- **Verifiable:** Produces easily-checkable certificates
- **Automatic:** No user interaction required

Applicable in practice

- **Implemented:** Resource Aware ML and Absynth
- **Effective:** Works for many typical programs
- **Efficient:** Inference via linear programming
Automatic Amortized Resource Analysis (AARA)

Type system for deriving symbolic resource bounds

‣ **Compositional:** Integrated with type systems or program logics
‣ **Expressive:** Bounds are multivariate resource polynomials
‣ **Reliable:** Formal soundness proof wrt. cost semantics
‣ **Verifiable:** Produces easily-checkable certificates
‣ **Automatic:** No user interaction required

Applicable in practice

‣ **Implemented:** Resource Aware ML and Absynth
‣ **Effective:** Works for many typical programs
‣ **Efficient:** Inference via linear programming

Type checking in linear time!
2. Shared (resource-aware) binary session types
Binary Session Types

- Implement message-passing concurrent programs
- Communication via typed bidirectional channels
- Curry-Howard correspondence with intuitionistic linear logic
- Client and provider have dual types

*Example type:* $\text{queue}_A = \&\{\text{ins} : A \rightarrow \text{queue}_A, \\
\quad \text{del} : \oplus\{\text{none} : 1, \\
\qquad \text{some} : A \otimes \text{queue}_A\}\}$
Binary Session Types

- Implement message-passing concurrent programs
- Communication via typed bidirectional channels
- Curry-Howard correspondence with intuitionistic linear logic
- Client and provider have dual types

**Example type:**

\[
\text{queue}_A = \&\{\text{ins} : A \to \text{queue}_A, \\
\text{del} : \oplus\{\text{none} : 1, \\
\text{some} : A \otimes \text{queue}_A\}\}
\]
Binary Session Types

- Implement message-passing concurrent programs
- Communication via typed bidirectional channels
- Curry-Howard correspondence with intuitionistic linear logic
- Client and provider have dual types

**Example type:**

\[
\text{queue}_A = \&\{ \text{ins} : A \rightarrow \text{queue}_A, \\
\text{del} : \oplus\{ \text{none} : 1, \\
\text{some} : A \otimes \text{queue}_A \} \}
\]

Receive msg of type A

Send msg of type A
Binary Session Types

- Implement message-passing concurrent programs
- Communication via typed bidirectional channels
- Curry-Howard correspondence with intuitionistic linear logic
- Client and provider have dual types

**Example type:**

\[ \text{queue}_A = & \{ \text{ins} : A \rightarrow \text{queue}_A, \]
\[ \text{del} : \oplus \{ \text{none} : 1, \]
\[ \text{some} : A \otimes \text{queue}_A \} \]
Example: Queue

queue_A = \&\{\text{ins} : A \rightarrow \text{queue}_A, \\
\quad \text{del} : \oplus\{\text{none} : 1, \\
\quad \quad \text{some} : A \otimes \text{queue}_A\}\}
Example: Queue

\[
\begin{align*}
(x : A) \ (t : \text{queue}_A) \vdash \text{elem} :: (s : \text{queue}_A) \\
\quad s & \leftarrow \text{elem} \leftarrow x \ t = \\
\quad \text{case } s \ (\text{ins} \Rightarrow y \leftarrow \text{recv } s ; \\
\quad t.\text{ins} ; \\
\quad \text{send } t \ y ; \\
\quad s \leftarrow \text{elem} \leftarrow x \ t \\
\quad | \ \text{del} \Rightarrow s.\text{some} ; \\
\quad \text{send } s \ x ; \\
\quad s \leftarrow t)
\end{align*}
\]

queue\(_A\) = \&\{\text{ins} : A \rightarrow \text{queue}_A, \\
\text{del} : \oplus\{\text{none} : 1, \\
\text{some} : A \otimes \text{queue}_A\}\}
Example: Queue

```
(x : A) (t : queue_A) ⊢ elem :: (s : queue_A)
  s ← elem ← x t =
  case s (ins ⇒ y ← recv s ;
          t.ins ;
          send t y ;
          s ← elem ← x t
         | del ⇒ s.some ;
          send s x ;
          s ← t)
```

queue_A = \&\{\text{ins : } A \rightarrow queue_A,\
          \text{del : } \oplus\{\text{none : } 1,\
                     \text{some : } A \otimes queue_A\}\}

recv 'ins' and y
send 'ins' and y
Example: Queue

\[(x : A) \ (t : queue_A) \vdash elem :: (s : queue_A)\]

\[s \leftarrow elem \leftarrow x \ t =\]

\[\text{case } s \begin{cases} \text{ins } \Rightarrow y \leftarrow \text{recv } s; \\
\quad t.\text{ins}; \\
\quad \text{send } t \ y; \\
\quad s \leftarrow elem \leftarrow x \ t \\
\end{cases}\]

\| del \Rightarrow s.\text{some}; \\
\quad \text{send } s \ x; \\
\quad s \leftarrow t\]

queue\_A = &\{\text{ins} : A \rightarrow queue\_A, \\
\text{del} : \oplus\{\text{none} : 1, \\
\text{some} : A \otimes queue\_A\}\}
Example: Queue

\[
(x : A) \ (t : \text{queue}_A) \vdash \text{elem} :: (s : \text{queue}_A) \\
\text{s} \leftarrow \text{elem} \leftarrow x \ t = \\
\text{case } s \text{ (ins } \Rightarrow y \leftarrow \text{recv } s ; \text{ t.ins ; send } t \ y ; \text{ send } s \ x ; \text{ send } s \ x ; \text{ send } s \ x ; \text{ send } s \ x ; s \leftarrow t) \\
\mid \text{del } \Rightarrow s.\text{some ; send } s \ x ; s \leftarrow t)
\]

\[
\text{queue}_A = \&\{\text{ins} : A \rightarrow \text{queue}_A, \text{del} : \oplus\{\text{none} : 1, \text{some} : A \otimes \text{queue}_A\}\}
\]
Example: Queue

\[
(x : A) \ (t : \text{queue}_A) \vdash \text{elem} :: (s : \text{queue}_A)
\]

\[
s \leftarrow \text{elem} \leftarrow x \ t = \\
\text{case } s \ (\text{ins} \Rightarrow y \leftarrow \text{recv } s \ ; \ t.\text{ins} ; \ \text{send } t \ y ; \ s \leftarrow \text{elem} \leftarrow x \ t \ | \ \text{del} \Rightarrow s.\text{some} ; \ \text{send } s \ x ; \ s \leftarrow t)
\]

queue\_A = \&\{\text{ins} : A \rightarrow \text{queue}_A, \ \\
\text{del} : \oplus\{\text{none} : 1, \ \text{some} : A \otimes \text{queue}_A\}\}
Example: Queue

Type checking in linear time!

\[
(x : A) \ (t : \text{queue}_A) \vdash \text{elem} :: (s : \text{queue}_A) \quad s \leftarrow \text{elem} \leftarrow x \ t = \\
\begin{cases}
\text{case } s \ (\text{ins} \Rightarrow y \leftarrow \text{recv } s ; \\
\hspace{1cm} t.\text{ins} ; \\
\hspace{1cm} \text{send } t \ y ; \\
\hspace{1cm} s \leftarrow \text{elem} \leftarrow x \ t) \\
| \text{del} \Rightarrow s.\text{some} ; \\
\hspace{1cm} \text{send } s \ x ; \\
\hspace{1cm} s \leftarrow t)
\end{cases}
\]

\[
\text{queue}_A = \&\{\text{ins} : A \rightarrow \text{queue}_A, \\
\text{del} : \oplus\{\text{none} : 1, \\
\text{some} : A \otimes \text{queue}_A\}\}
\]
Example: Auction

\[
\text{auction} = \oplus \{\text{running} : \& \{\text{bid} : \text{id} \supset \text{money} \rightarrow \text{auction}\}, \]
\[
\text{ended} : \& \{\text{collect} : \text{id} \supset \oplus \{\text{won} : \text{monalisa} \otimes \text{auction}, \}
\text{lost} : \text{money} \otimes \text{auction}\}\}
\]
Example: Auction

\[ \text{auction} = \bigoplus \{ \text{running} : \& \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \text{auction} \}, \]
\[ \text{ended} : \& \{ \text{collect} : \text{id} \supset \bigoplus \{ \text{won} : \text{monalisa} \otimes \text{auction}, \]
\[ \text{lost} : \text{money} \otimes \text{auction} \} \} \} \]
Example: Auction

\[
\text{auction} = \oplus \{ \text{running} : \& \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \text{auction} \}, \\
\text{ended} : \& \{ \text{collect} : \text{id} \supset \oplus \{ \text{won} : \text{monalisa} \otimes \text{auction}, \\
\text{lost} : \text{money} \otimes \text{auction} \} \} \}
\]
Example: Auction

sends status of auction

offers choice of bidding

receive id and money

\[ \text{auction} = \oplus \{ \text{running} : & \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \text{auction} \}, \]
\[ \text{ended} : \& \{ \text{collect} : \text{id} \supset \oplus \{ \text{won} : \text{monalisa} \otimes \text{auction}, \}
\[ \text{lost} : \text{money} \otimes \text{auction} \} \} \} \]
Example: Auction

\[
auction = \oplus \{ \text{running} : \land \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \text{auction} \}, \\
\text{ended} : \land \{ \text{collect} : \text{id} \supset \oplus \{ \text{won} : \text{monalisa} \otimes \text{auction}, \\
\text{lost} : \text{money} \otimes \text{auction} \} \} \}
\]
Example: Auction

\[
auction = \oplus \{ \text{running} : \& \{ \text{bid} : \text{id} \supset \text{money} \rightarrow auction \}, \\
\quad \text{ended} : \& \{ \text{collect} : \text{id} \supset \oplus \{ \text{won} : \text{monalisa} \times auction, \\
\quad \quad \text{lost} : \text{money} \times auction \} \} \} 
\]
Example: Auction

\[
auction = \oplus \{ \text{running} : \& \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \text{auction} \}, \\
\text{ended} : \& \{ \text{collect} : \text{id} \supset \oplus \{ \text{won} : \text{monalisa} \otimes \text{auction}, \\
\text{lost} : \text{money} \otimes \text{auction} \} \} \}
\]
Example: Auction

\[
auction = \oplus \{ \text{running} : \& \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \text{auction} \}, \text{ended} : \& \{ \text{collect} : \text{id} \supset \oplus \{ \text{won} : \text{monalisa} \otimes \text{auction}, \text{lost} : \text{money} \otimes \text{auction} \} \} \}
\]
Example: Auction

\[
auction = \oplus \{ \text{running} : \& \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \text{auction} \},
\]
\[
\text{ended} : \& \{ \text{collect} : \text{id} \supset \oplus \{ \text{won} : \text{monalisa} \otimes \text{auction},
\text{lost} : \text{money} \otimes \text{auction} \} \} \}
\]

- sends status of auction
- offers choice of bidding
- receive id and money
- recurse
- offers choice to collect
- sends result of bidding
- send Mona Lisa
- send back money
Resource-Aware Session Types

- Each process *stores potential* in functional data
- Potential can be *transferred via messages*
- Potential is used to *pay for performed work*
Resource-Aware Session Types

- Each *process stores potential* in functional data
- Potential can be *transferred via messages*
- Potential is used to *pay for performed work*

Potential transfer only at the type level, not at runtime.
Resource-Aware Session Types

- Each process stores potential in functional data
- Potential can be transferred via messages
- Potential is used to pay for performed work

Potential transfer only at the type level, not at runtime.

User-defined cost metric.
Resource-Aware Session Types

- Each process stores potential in functional data
- Potential can be transferred via messages
- Potential is used to pay for performed work
- Message potential is a function of (functional) payload

\[ A, B, C ::= \tau \supset A \]

Input value of type \( \tau \) and continue as \( A \)

\[ \tau \land A \]

Output value of type \( \tau \) and continue as \( A \)

\[ \underline{2} (\text{int}) \]

User-defined cost metric.

Potential transfer only at the type level, not at runtime.
Resource-Aware Session Types

- Each **process stores potential** in functional data
- Potential can be **transferred via messages**
- Potential is used to **pay for performed work**

- Message potential is a function of (functional) payload

\[ A, B, C ::= \tau \supset A \quad \tau \land A \]

- Syntactic sugar (no payload)

\[ A ::= \ldots | \triangleright^r A | \triangleleft^r A \]

- Only in intermediate language:

\[
\begin{align*}
&\text{get } x_m \{r\} ; P \\
&\text{pay } x_m \{r\} ; P
\end{align*}
\]
Resource-Aware Session Types

- Each process stores potential in functional data
- Potential can be transferred via messages
- Potential is used to pay for performed work

Message potential is a function of (functional) payload

\[
A, B, C ::= \tau \triangleright A \\
\tau \land A
\]

Syntactic sugar (no payload)

\[
A ::= \ldots | \triangleright^r A | \triangleleft^r A
\]

Only in intermediate language:

\[
\text{get } x_m \{r\} ; P \\
\text{pay } x_m \{r\} ; P
\]
Example: Type of an Auction Contract
Sharing: Need to acquire contract before use.

\[
auction = \uparrow^S_L \perp^{11} \oplus \{\text{running} : \&\{\text{bid} : \text{id} \bowtie \text{money} \rightarrow \triangleright^1 \downarrow^S_L \text{auction}, \\
\text{cancel} : \triangleright^8 \downarrow^S_L \text{auction}\}, \\
\text{ended} : \&\{\text{collect} : \text{id} \bowtie \\
\oplus\{\text{won} : \text{lot} \otimes \triangleright^3 \downarrow^S_L \text{auction}, \\
\text{lost} : \text{money} \otimes \downarrow^S_L \text{auction}\}, \\
\text{cancel} : \triangleright^8 \downarrow^S_L \text{auction}\}\}
\]

Example: Type of an Auction Contract
Example: Type of an Auction Contract

\[
auction = \uparrow^S \downarrow^L \oplus \{\text{running} : \&\{\text{bid} : \text{id} \supset \text{money} \rightarrow \Rightarrow^1 \downarrow^S \text{auction}, \\
\text{cancel} : \Rightarrow^8 \downarrow^S \text{auction}\}, \\
\text{ended} : \&\{\text{collect} : \text{id} \supset \\
\oplus\{\text{won} : \text{lot} \otimes \Rightarrow^3 \downarrow^S \text{auction}, \\
\text{lost} : \text{money} \otimes \downarrow^S \text{auction}\}, \\
\text{cancel} : \Rightarrow^8 \downarrow^S \text{auction}\}\}
\]
This article illustrates the key concepts and functionality of (non-linear) bound inference to ordinary contracts. This section uses a simple auction contract to develop the auction protocol. To leave the logical foundation intact, the integration of a linear type system would accommodate the explicit notions of automatic amortized resource analysis (AARA) to accommodate the explicit notions of automatic amortized resource analysis (AARA).

We prove the type soundness of Nomos with respect to the highest bidder receives the lot while all other bidders.

We integrate linear session types that support controlled exclusion, the session type demarcates the parts of the process expressions. Since there exist multiple bidders in an auction, we use a shared auction session and its end. Programmatically, the highest bidder receives the lot while all other bidders.

Nomos ses

The critical section

Example: Type of an Auction Contract

\[
\text{auction} = \uparrow^{S}_{L} \overset{11}{\triangleleft} \text{running} \& \{\text{bid} : \text{id} \triangleright \text{money} \rightarrow \downarrow^{1}_{L} \text{auction}, \\
\text{cancel} : \downarrow^{8}_{L} \text{auction} \}, \\
\text{ended} : \& \{\text{collect} : \text{id} \triangleright \\
\text{\Theta}\{\text{won} : \text{lot} \otimes \downarrow^{3}_{L} \text{auction}, \\
\text{lost} : \text{money} \otimes \downarrow^{S}_{L} \text{auction} \}, \\
\text{cancel} : \downarrow^{8}_{L} \text{auction} \}
\]
Example: Type of an Auction Contract

\[
\text{auction} = \uparrow_{\downarrow}^{\uparrow} \oplus \{ \text{running} : \&\{ \text{bid} : \text{id} \triangleright \text{money} \rightarrow \downarrow_{\downarrow}^{\uparrow} \text{auction}, \\
\text{cancel} : \uparrow \downarrow_{\downarrow}^{\uparrow} \text{auction} \}, \\
\text{ended} : \&\{ \text{collect} : \text{id} \triangleright \\
\quad \oplus\{ \text{won} : \text{lot} \otimes \downarrow_{\downarrow}^{\uparrow} \text{auction}, \\
\quad \text{lost} : \text{money} \otimes \downarrow_{\downarrow}^{\uparrow} \text{auction} \}, \\
\quad \text{cancel} : \uparrow \downarrow_{\downarrow}^{\uparrow} \text{auction} \} \}
\]

Sending a functional value.

Action can be open (running) or closed (ended).
Example: Type of an Auction Contract

\[
\text{auction} = \uparrow^S_L \triangleleft^{11} \oplus \{ \text{running} : \&\{ \text{bid} : \text{id} \supset \text{money} \rightarrow \triangleright^1 \downarrow^S_L \text{auction}, \\
\text{cancel} : \triangleright^8 \downarrow^S_L \text{auction} \}, \\
\text{ended} : \&\{ \text{collect} : \text{id} \supset \\
\oplus\{ \text{won} : \text{lot} \otimes \triangleright^3 \downarrow^S_L \text{auction}, \\
\text{lost} : \text{money} \otimes \downarrow^S_L \text{auction} \}, \\
\text{cancel} : \triangleright^8 \downarrow^S_L \text{auction} \}\}
\]
Example: Type of an Auction Contract

\[
auction = \uparrow^{S}_L \downarrow^{11} \oplus \{\text{running} : \&\{\text{bid} : \text{id} \triangleright \text{money} \rightarrow \downarrow^{1}_L \text{auction}, \\
\text{cancel} : \downarrow^{8}_L \text{auction}\}, \\
\text{ended} : \&\{\text{collect} : \text{id} \triangleright \\
\oplus\{\text{won} : \text{lot} \otimes \downarrow^{3}_L \text{auction}, \\
\text{lost} : \text{money} \otimes \downarrow^{S}_L \text{auction}\}, \\
\text{cancel} : \downarrow^{8}_L \text{auction}\}\}
\]
auction = \mathcal{S} \triangleright_{\mathcal{L}}^{11} \oplus \{ \text{running} : \& \{ \text{bid} : \text{id} \triangleright \text{money} \rightarrow \triangleright^{1}_{\mathcal{L}} \mathcal{S} \text{ auction}, \text{cancel} : \triangleright^{8}_{\mathcal{L}} \mathcal{S} \text{ auction}\}, \text{ended} : \& \{ \text{collect} : \text{id} \triangleright \}
\oplus \{ \text{won} : \text{lot} \otimes \triangleright^{3}_{\mathcal{L}} \mathcal{S} \text{ auction}, \text{lost} : \text{money} \otimes \triangleright^{8}_{\mathcal{L}} \mathcal{S} \text{ auction}\}, \text{cancel} : \triangleright^{8}_{\mathcal{L}} \mathcal{S} \text{ auction}\}\}

Example: Type of an Auction Contract
At the beginning, you have to pay 11 units to cover the worst-case gas cost.

\[
auction = \uparrow^S_L \downarrow^{11}_L \oplus \{\text{running}: \&\{\text{bid}: \text{id} \supset \text{money} \rightarrow \triangleright^1 \downarrow^S_L \text{auction}, \\
\text{cancel}: \triangleright^8 \downarrow^S_L \text{auction}\}, \\
\text{ended}: \&\{\text{collect}: \text{id} \supset \\
\oplus\{\text{won}: \text{lot} \otimes \triangleright^3 \downarrow^S_L \text{auction}, \\
\text{lost}: \text{money} \otimes \downarrow^S_L \text{auction}\}, \\
\text{cancel}: \triangleright^8 \downarrow^S_L \text{auction}\}\}
\]

Example: Type of an Auction Contract
At the beginning, you have to pay 11 units to cover the worst-case gas cost.

\[
auction = \uparrow_{L}^{S} \downarrow^{11}_{L} \oplus \{\text{running} : \&\{\text{bid} : \text{id} \supset \text{money} \rightarrow \uparrow^{1}_{L} \downarrow_{L}^{S} \text{auction}, \\
\text{cancel} : \uparrow^{8}_{L} \downarrow_{L}^{S} \text{auction}\}, \\
\text{ended} : \&\{\text{collect} : \text{id} \supset \\
\oplus\{\text{won} : \text{lot} \otimes \uparrow^{3}_{L} \downarrow_{L}^{S} \text{auction}, \\
\text{lost} : \text{money} \otimes \downarrow_{L}^{S} \text{auction}\}, \\
\text{cancel} : \uparrow^{8}_{L} \downarrow_{L}^{S} \text{auction}\}\}
\]

Gas cost is given by a cost semantics and the type system ensures 11 is the worst-case.

Example: Type of an Auction Contract
At the beginning, you have to pay 11 units to cover the worst-case gas cost.

\[
auction = \uparrow^S_L \triangleleft^{11} \oplus \{\text{running} : \&\{\text{bid} : \text{id} \supset \text{money} \rightarrow \triangleright^1 \downarrow^S_L \text{auction},
\text{cancel} : \triangleright^8 \downarrow^S_L \text{auction}\},
\text{ended} : \&\{\text{collect} : \text{id} \supset
\oplus\{\text{won} : \text{lot} \otimes \triangleright^3 \downarrow^S_L \text{auction},
\text{lost} : \text{money} \otimes \downarrow^S_L \text{auction}\},
\text{cancel} : \triangleright^8 \downarrow^S_L \text{auction}\}\}
\]

If the worst-case path is not taken then the leftover is returned.

Gas cost is given by a cost semantics and the type system ensures 11 is the worst-case.

Example: Type of an Auction Contract
At the beginning, you have to pay 11 units to cover the worst-case gas cost.

\[
\text{auction} = \uparrow^S_L \uparrow^{11} \oplus \{ \text{running} : \&\{ \text{bid} : \text{id} \supset \text{money} \rightarrow \uparrow^1 \downarrow^S_L \text{auction}, \\
\text{cancel} : \uparrow^8 \downarrow^S_L \text{auction} \}, \\
\text{ended} : \&\{ \text{collect} : \text{id} \supset \\
\oplus\{ \text{won} : \text{lot} \otimes \uparrow^3 \downarrow^S_L \text{auction}, \\
\text{lost} : \text{money} \otimes \downarrow^S_L \text{auction} \}, \\
\text{cancel} : \uparrow^8 \downarrow^S_L \text{auction} \} \}
\]

If the worst-case path is not taken then the leftover is returned.

Gas cost is given by a cost semantics and the type system ensures 11 is the worst-case.

Example: Type of an Auction Contract
Implementation of a Running Auction

\[\text{auction} = \uparrow^{S}_{L} \downarrow^{11} \oplus \{\text{running : } \&\{\text{bid : id } \supset \text{money } \rightarrow \downarrow^{1}_{L} S_{\text{auction}}\},\]

\[(b : \text{bids}) ; (M : \text{money}), (ml : \text{monalisa}) \vdash \text{run :: (sa : auction)}\]

\[sa \leftarrow \text{run} \; b \leftarrow M \; l = \]
\[la \leftarrow \text{accept} \; sa ;\]
\[la.\text{running} ;\]
\[\text{case } la\]
\[\text{(bid } \Rightarrow r \leftarrow \text{recv} \; la ;\]
\[m \leftarrow \text{recv} \; la ;\]
\[sa \leftarrow \text{detach} \; la ;\]
\[m.\text{value} ;\]
\[v \leftarrow \text{recv} \; m ;\]
\[b' = \text{addbid} \; b \; (r, v) ;\]
\[M' \leftarrow \text{add} \leftarrow M \; m ;\]
\[sa \leftarrow \text{run} \; b' \leftarrow M' \; ml)\]
Implementation of a Running Auction

\[
\text{auction} = \uparrow_S L \downarrow^{11} \oplus \{\text{running} : \& \{\text{bid} : \text{id} \supset \text{money} \rightarrow \uparrow^1 L \downarrow^S \text{auction}\},
\]

\( (b : \text{bids}) ; (M : \text{money}), (ml : \text{monalisa}) \vdash \text{run} :: (sa : \text{auction}) \)

\( sa \leftarrow \text{run} b \leftarrow M l = \)

\( la \leftarrow \text{accept} \text{sa} ; \)

\( la.\text{running} ; \)

\( \text{case} \ la \)

\( (\text{bid} \Rightarrow r \leftarrow \text{recv} la ; \)

\( m \leftarrow \text{recv} la ; \)

\( sa \leftarrow \text{detach} la ; \)

\( m.\text{value} ; \)

\( v \leftarrow \text{recv} m ; \)

\( b' = \text{addbid} b (r, v) ; \)

\( M' \leftarrow \text{add} \leftarrow M m ; \)

\( sa \leftarrow \text{run} b' \leftarrow M' ml) \)
Implementation of a Running Auction

\[
\text{auction} = \uparrow^S_L \downarrow^{11} \ominus \{ \text{running} : \& \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \uparrow^1 \downarrow^S_L \text{auction} \},
\]

\[
(b : \text{bids}) ; (M : \text{money}), (ml : \text{monalisa}) \vdash \text{run} :: (sa : \text{auction})
\]

\[
sa \leftarrow \text{run} \ b \leftarrow M \ l =
\]

\[
dl \leftarrow \text{accept} \ sa ;
\]

\[
dl.\text{running}
\]

\[
\text{case} \ \ndl
\]

\[
(b \Rightarrow r \leftarrow \text{recv} \ \ndl ;
\]

\[
m \leftarrow \text{recv} \ \ndl ;
\]

\[
sa \leftarrow \text{detach} \ \ndl ;
\]

\[
m.\text{value} ;
\]

\[
v \leftarrow \text{recv} \ m ;
\]

\[
b' = \text{addbid} \ b (r, v) ;
\]

\[
M' \leftarrow \text{add} \leftarrow M \ m ;
\]

\[
sa \leftarrow \text{run} \ b' \leftarrow M' \ ml)
\]
Implementation of a Running Auction

\[
\text{auction} = \uparrow^S_L \downarrow^{11}\oplus \{\text{running} : \land \{\text{bid} : \text{id} \supset \text{money} \rightarrow \downarrow^1_L \uparrow^S_L \text{auction}\},
\]

(b : bids) ; (M : money), (ml : monalisa) \vdash \text{run} :: (sa : auction)

\[
\text{sa} \leftarrow \text{run} \ b \leftarrow M \ l =
\]

\[
\begin{align*}
\text{la} & \leftarrow \text{accept} \ sa ; \\
\text{la} & . \text{running} ; \\
\text{case} \ \text{la} \\
\text{(bid} \Rightarrow r & \leftarrow \text{recv} \ l a ; \\
\text{m} & \leftarrow \text{recv} \ l a ; \\
\text{sa} & \leftarrow \text{detach} \ l a ; \\
\text{m} & . \text{value} ; \\
\text{v} & \leftarrow \text{recv} \ m ; \\
\text{b'} & = \text{addbid} \ b \ (r,v) ; \\
\text{M'} & \leftarrow \text{add} \leftarrow M \ m ; \\
\text{sa} & \leftarrow \text{run} \ b' \leftarrow M' \ ml)
\end{align*}
\]
Implementation of a Running Auction

\[
\text{auction} = \uparrow_{\mathbf{L}}^{S} \ominus \{\text{running} : \&\{\text{bid} : \text{id} \supset \text{money} \rightarrow \downarrow_{\mathbf{L}}^{1} S\text{auction}\},
\]

\([b : \text{bids}] ; (M : \text{money}), (ml : \text{monalisa}) \vdash \text{run} :: (sa : \text{auction}) \]

\[
sa \leftarrow \text{run} \ b \leftarrow M \ l = \\
l \ a \leftarrow \text{accept} \ sa ; \\
\text{la.running} ; \\
\text{case} \ la
\]

\[
(b \Rightarrow r \leftarrow \text{recv} \ la ; \\
m \leftarrow \text{recv} \ la ; \\
\text{sa} \leftarrow \text{detach} \ la ; \\
m \text{.value} ; \\
v \leftarrow \text{recv} \ m ; \\
b' = \text{addbid} \ b \ (r, v) ; \\
M' \leftarrow \text{add} \leftarrow M \ m ; \\
\text{sa} \leftarrow \text{run} \ b' \leftarrow M' \ ml)
\]

- accept ‘acquire’ (\(\uparrow_{\mathbf{L}}^{S}\))
- send status ‘running’
- recv ‘id’ and ‘money’
- detach from client (\(\downarrow_{\mathbf{L}}^{S}\))
Implementation of a Running Auction

\[ \text{auction} = \uparrow^{S}_{L} \{ \text{running} : \& \{ \text{bid : id} \supset \text{money} \rightarrow \downarrow^{1}_{L} \} \}, \]

\((b : \text{bids}) ; (M : \text{money}), (ml : \text{monalisa}) \vdash \text{run} :: (sa : \text{auction})\]

\[
\begin{align*}
sa & \leftarrow \text{run} b \leftarrow M l = \\
& \quad la \leftarrow \text{accept} sa ; \\
& \quad \text{la.running} ; \\
& \quad \text{case la} \\
& \quad \left( \text{bid} \Rightarrow r \leftarrow \text{recv} la ; \\
& \quad \quad m \leftarrow \text{recv} la ; \\
& \quad \quad sa \leftarrow \text{detach} la ; \\
& \quad \quad m.\text{value} ; \\
& \quad \quad v \leftarrow \text{recv} m ; \\
& \quad \quad b' = \text{addbid} b (r, v) ; \\
& \quad \quad M' \leftarrow \text{add} \leftarrow M m ; \\
& \quad \quad sa \leftarrow \text{run} b' \leftarrow M' ml)
\end{align*}
\]

- accept ‘acquire’ \((\uparrow^{S}_{L})\)
- send status ‘running’
- recv ‘id’ and ‘money’
- detach from client \((\downarrow^{S}_{L})\)
- add bid and money
Implementation of a Running Auction

\[ \text{auction} = \uparrow^S_L \odot \{ \text{running} : \land \{ \text{bid} : \text{id} \supset \text{money} \rightarrow \downarrow^1_L \text{auction} \}, \]

(b : bids); (M : money), (ml : monalisa) \vdash \text{run} :: (sa : auction)

\[ \text{sa} \leftarrow \text{run} \ b \leftarrow M \ l = \]

la \leftarrow \text{accept} \ sa ;

la.running ;

case la

(bid \Rightarrow r \leftarrow \text{recv} \ la ;

m \leftarrow \text{recv} \ la ;

sa \leftarrow \text{detach} \ la ;

m.value ;

v \leftarrow \text{recv} \ m ;

b' = \text{addbid} \ b \ (r, v) ;

M' \leftarrow \text{add} \leftarrow M \ m ;

sa \leftarrow \text{run} \ b' \leftarrow M' \ ml) \]

- accept 'acquire' (\uparrow^S_L)
- send status 'running'
- recv 'id' and 'money'
- detach from client (\downarrow^S_L)
- add bid and money
- no work constructs!
How to Use the Potential

**Payment schemes (amortized cost)**
- Ensure constant gas cost in the presence of costly operations
- Overcharge for cheap operations and store gas in contract
- Similar to storing ether in memory in EVM but part of contract

**Explicit gas bounds**
- Add an additional argument that carries potential
- User arg $N \sim$ maximal number of players $\Rightarrow$ gas bound is $81N + 28$

**Enforce constant gas cost**
- Simply disable potential in contract state
- Require messages to only carry constant potential
Computation on a Blockchain

**Blockchain state:** shared processes waiting to be acquired

\[ \text{contr}_1(\bar{u}_1, \bar{v}_1) \quad \text{contr}_2(\bar{u}_2, \bar{v}_2) \quad \ldots \quad \text{contr}_n(\bar{u}_n, \bar{v}_n) \]
Computation on a Blockchain

**Blockchain state:** shared processes waiting to be acquired

Contracts store functional and linear data.

\[
\text{contr}_1(\tilde{u}_1, \tilde{v}_1) \quad \text{contr}_2(\tilde{u}_2, \tilde{v}_2) \quad \ldots \quad \text{contr}_n(\tilde{u}_n, \tilde{v}_n)
\]
Computation on a Blockchain

**Blockchain state:** shared processes waiting to be acquired

\[
\text{contr}_1(\vec{u}_1, \vec{v}_1) \quad \text{contr}_2(\vec{u}_2, \vec{v}_2) \quad \ldots \quad \text{contr}_n(\vec{u}_n, \vec{v}_n)
\]

Contracts store functional and linear data.

Channel name = address
Computation on a Blockchain

**Blockchain state:** shared processes waiting to be acquired

\[ \text{contr}_1(\bar{u}_1, \bar{v}_1) \quad \text{contr}_2(\bar{u}_2, \bar{v}_2) \quad \cdots \quad \text{contr}_n(\bar{u}_n, \bar{v}_n) \]

Contracts store functional and linear data.

Channel name = address

**Transaction:** client submits code of a linear process

\[ \text{contr}_1(\bar{u}_1, \bar{v}_1) \quad \text{contr}_2(\bar{u}_2, \bar{v}_2) \quad \cdots \quad \text{contr}_n(\bar{u}_n, \bar{v}_n) \quad \text{client} \]
Computation on a Blockchain

**Blockchain state:** shared processes waiting to be acquired

\[
\text{contr}_1(u_1, v_1) \quad \text{contr}_2(u_2, v_2) \quad \ldots \quad \text{contr}_n(u_n, v_n)
\]

- Client process can acquire existing contracts
- Client process can spawn new (shared) processes -> new contracts
- Client process needs to terminates in a new valid state

**Transaction:** client submits code of a linear process

\[
\text{contr}_1(u_1, v_1) \quad \text{contr}_2(u_2, v_2) \quad \ldots \quad \text{contr}_n(u_n, v_n) \quad \text{client}
\]

Contracts store functional and linear data.

Channel name = address
Computation on a Blockchain

**Blockchain state:** shared processes waiting to be acquired

\[
\begin{align*}
\text{contr}_1(\bar{u}_1, \bar{v}_1) & \quad \text{c}_1 \\
\text{contr}_2(\bar{u}_2, \bar{v}_2) & \quad \text{c}_2 \\
\vdots & \quad \vdots \\
\text{contr}_n(\bar{u}_n, \bar{v}_n) & \quad \text{c}_n
\end{align*}
\]

Contracts store functional and linear data.

Channel name = address

**Transaction:** client submits code of a linear process

\[
\begin{align*}
\text{contr}_1(\bar{u}_1, \bar{v}_1) & \quad \text{c}_1 \\
\text{contr}_2(\bar{u}_2, \bar{v}_2) & \quad \text{c}_2 \\
\vdots & \quad \vdots \\
\text{contr}_n(\bar{u}_n, \bar{v}_n) & \quad \text{c}_n \\
\text{client}
\end{align*}
\]

- Client process can acquire existing contracts
- Client process can spawn new (shared) processes \( \rightarrow \) new contracts
- Client process needs to terminates in a new valid state

Contract should have default clients.
Blockchain, Type Checking, and Verification

**Type checking is part of the attack surface**

- Contract code can be checked at publication time
- User code needs to be checked for each transaction
- Denial of service attacks are possible
- *Nomos type checking is linear in the size of the program*

**Verification of Nomos program is possible**

- Dynamic semantics specifies runtime behavior
- Directly applicable to verification in Coq
- Nomos’ type system guarantees some important properties
Nomos

A statically-typed, strict, functional language for digital contracts

• Automatic amortized resource analysis for static gas bounds
• Shared binary session types for transparent & safe contract interfaces
• Linear type system for accurately reflecting assets

References

• POPL ’17: AARA for OCaml (RaML)  • arXiv ’19: Nomos
• LICS ’18: Resource-Aware Session Types

Ongoing work: implementation

• Parser ✓  • Type checker ✓  • Interpreter  • Compiler
Nomos

A statically-typed, strict, functional language for digital contracts

• Automatic amortized resource analysis for static gas bounds
• Shared binary session types for transparent & safe contract interfaces
• Linear type system for accurately reflecting assets

References

• POPL ’17: AARA for OCaml (RaML)
• LICS ’18: Resource-Aware Session Types
• arXiv ’19: Nomos

Ongoing work: implementation

• Parser ✓
• Type checker ✓
• Interpreter
• Compiler