Programming language developed at Carnegie Mellon

Nomos: Resource-Aware Session Types for Programming Digital Contracts

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

With some slides from Ankush.

Carnegie Mellon University

ETH 2019

Digital Contracts (or Smart Contracts)

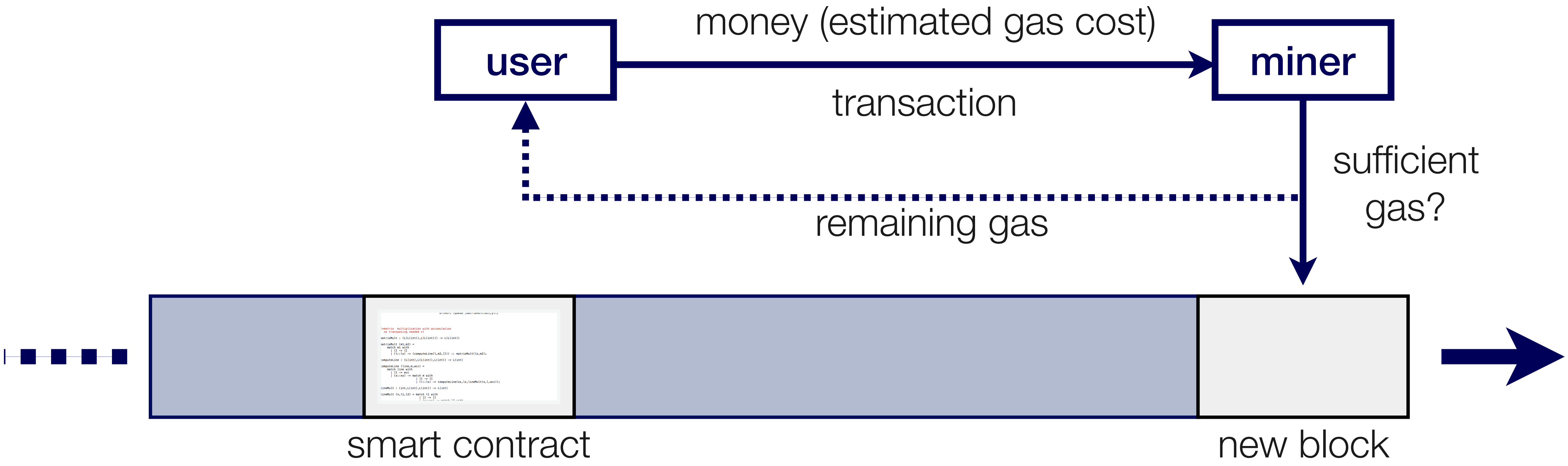
Smart contracts (Ethereum): programs stored on a blockchain

- 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*



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

KLINT FINLEY BUSINESS 06.18.16 04:30 AM

A $50 MILLION HACK JUST SHOWED THAT THE DAO WAS ALL TOO HUMAN

'$300m in cryptocurrency' accidentally lost forever due to bug

User mistakenly takes control of hundreds of wallets containing cryptocurrency Ether, destroying them in a panic while trying to give

CRYTPO ENTOMOLOGY

A coding error led to $30 million in ethereum being stolen

Can Programming Languages Prevent Bugs?

Can Programming Languages Prevent Bugs?

Yes!

Can Programming Languages Prevent Bugs?

Yes!

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, …)

Can Programming Languages Prevent Bugs?

Yes!

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

Can Programming Languages Prevent Bugs?

Yes!

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, …)

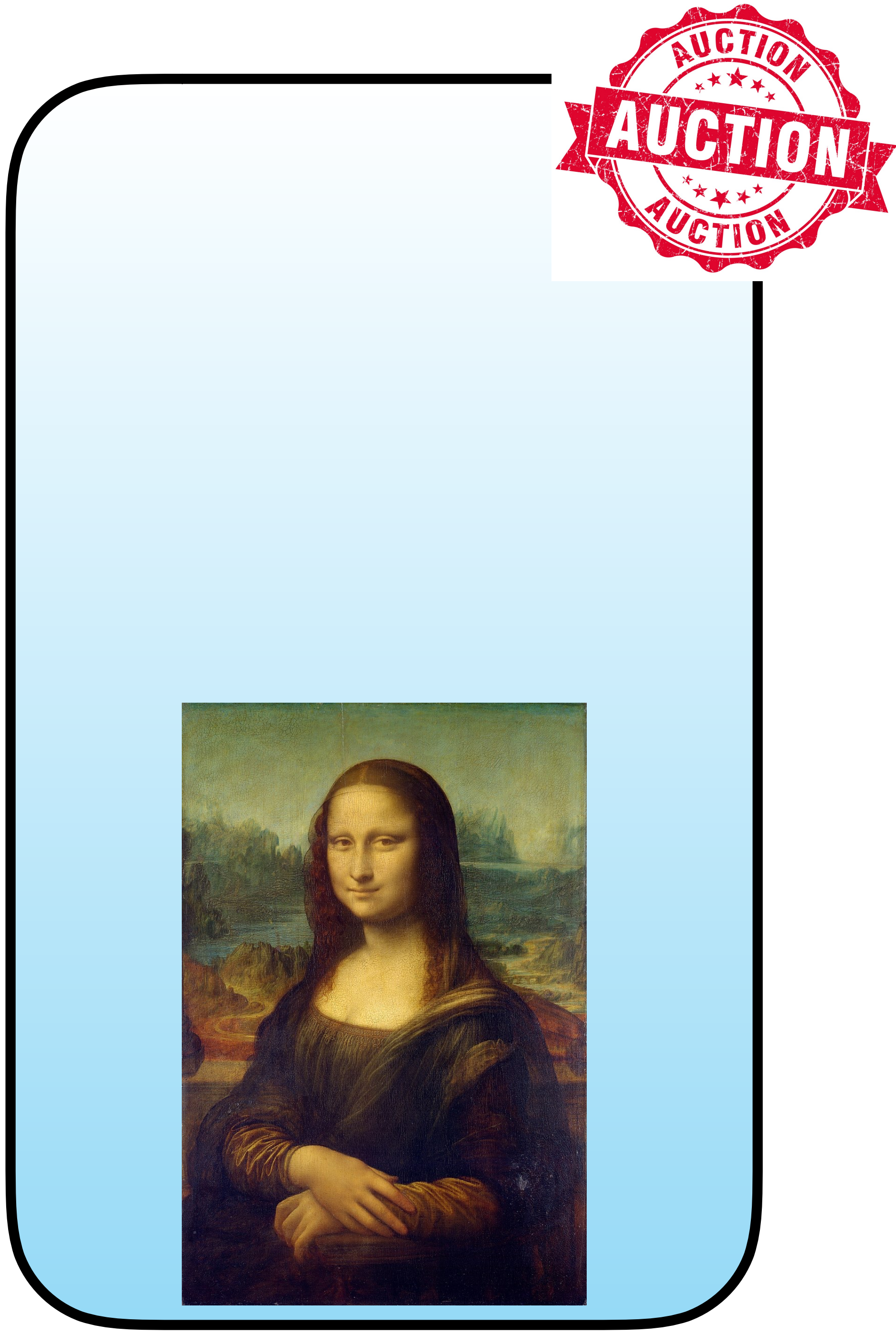
Languages for Digital Contracts

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

Nomos

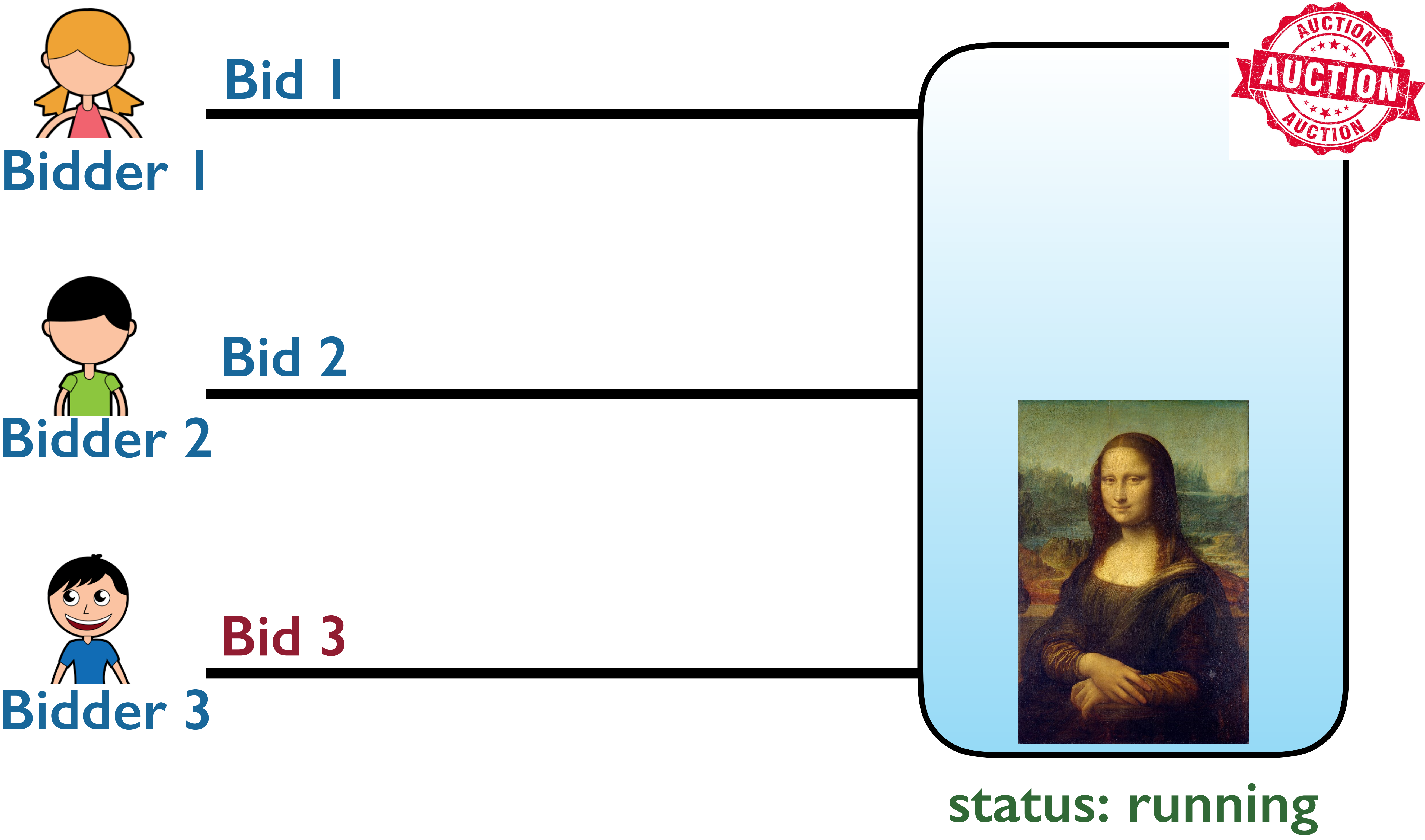
- Build on state-of-the art: statically-typed, strict, functional language
- Address domain-specific issues

Domain Specific Bugs: Auction Contract



status: running

Domain Specific Bugs: Auction Contract



Bidder 1

Bid 1

Bidder 2

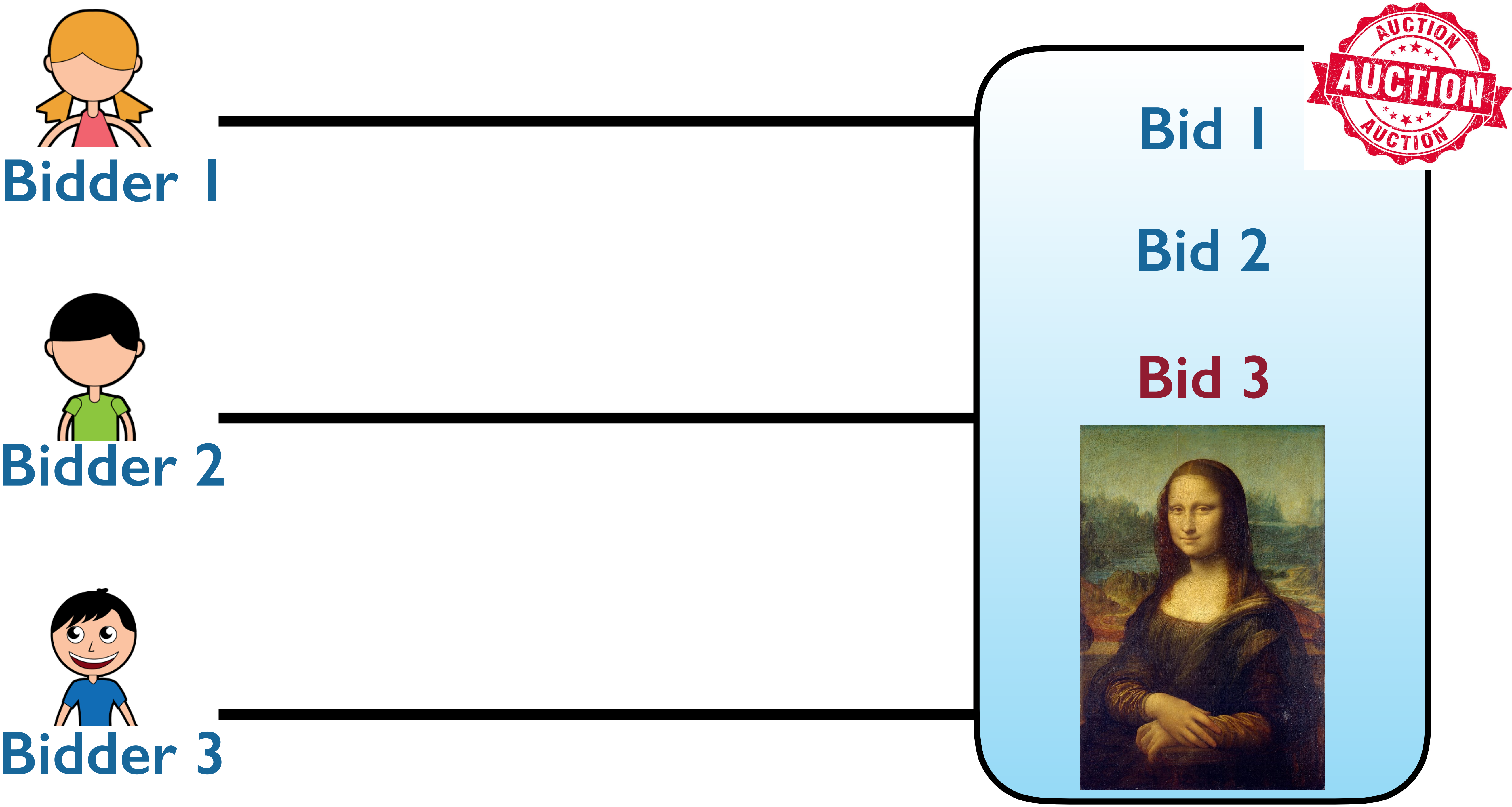
Bid 2

Bidder 3

Bid 3

status: running

Domain Specific Bugs: Auction Contract



Bidder 1

Bidder 2

Bidder 3

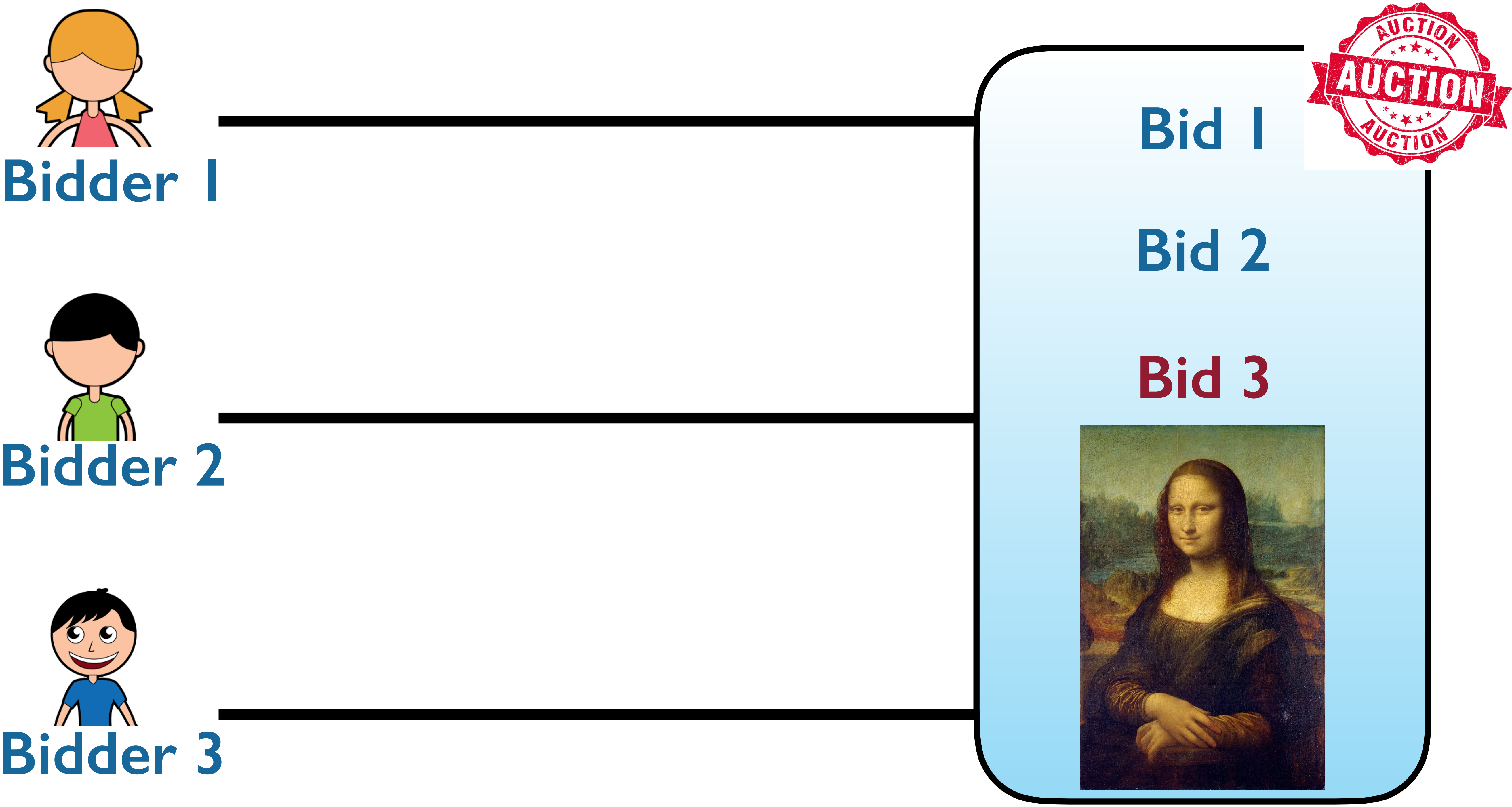
Bid 1

Bid 2

Bid 3

status: running

Domain Specific Bugs: Auction Contract



Bidder 1

Bidder 2

Bidder 3

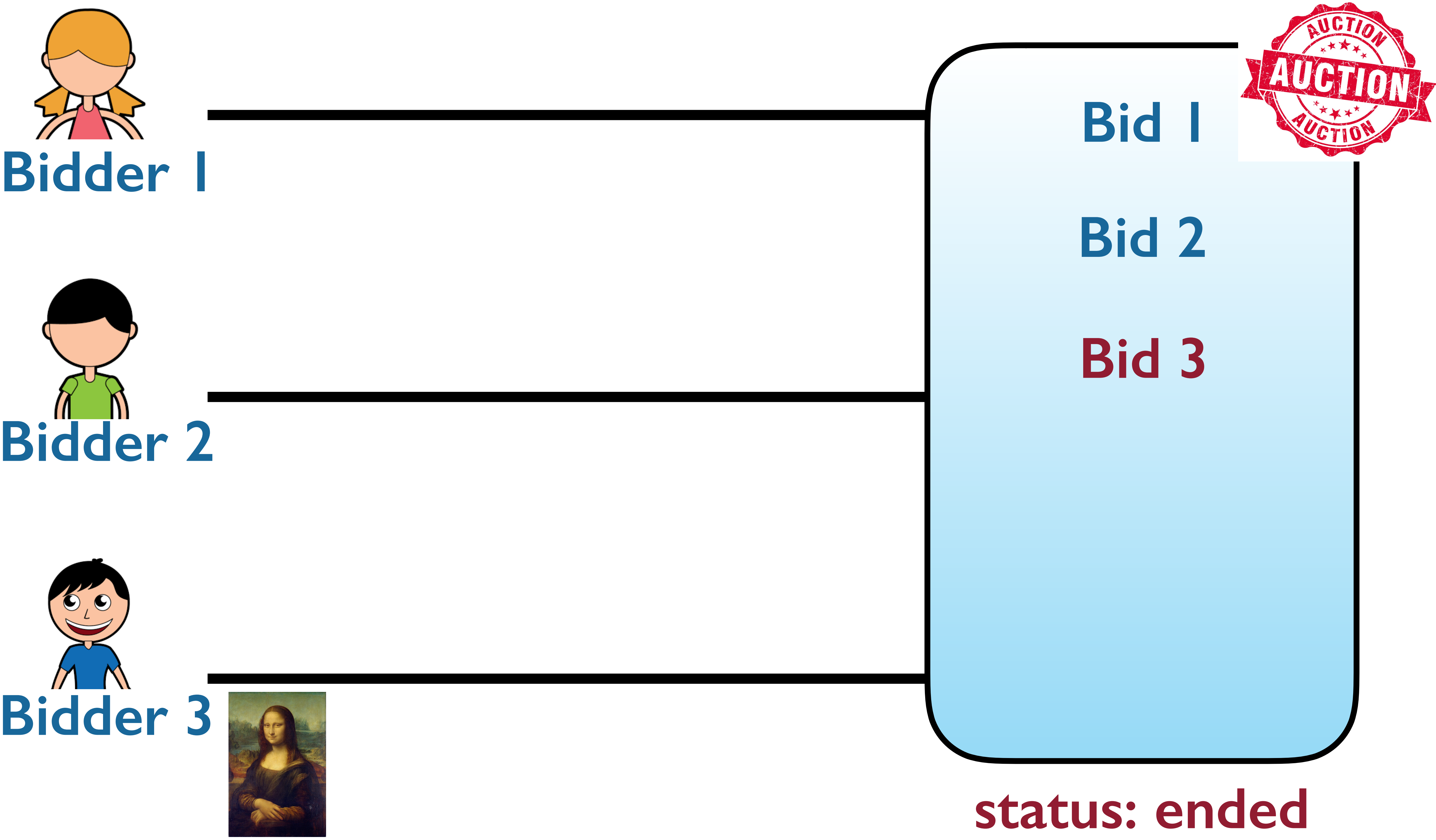
Bid 1

Bid 2

Bid 3

status: ended

Domain Specific Bugs: Auction Contract



Bidder 1

Bidder 2

Bidder 3

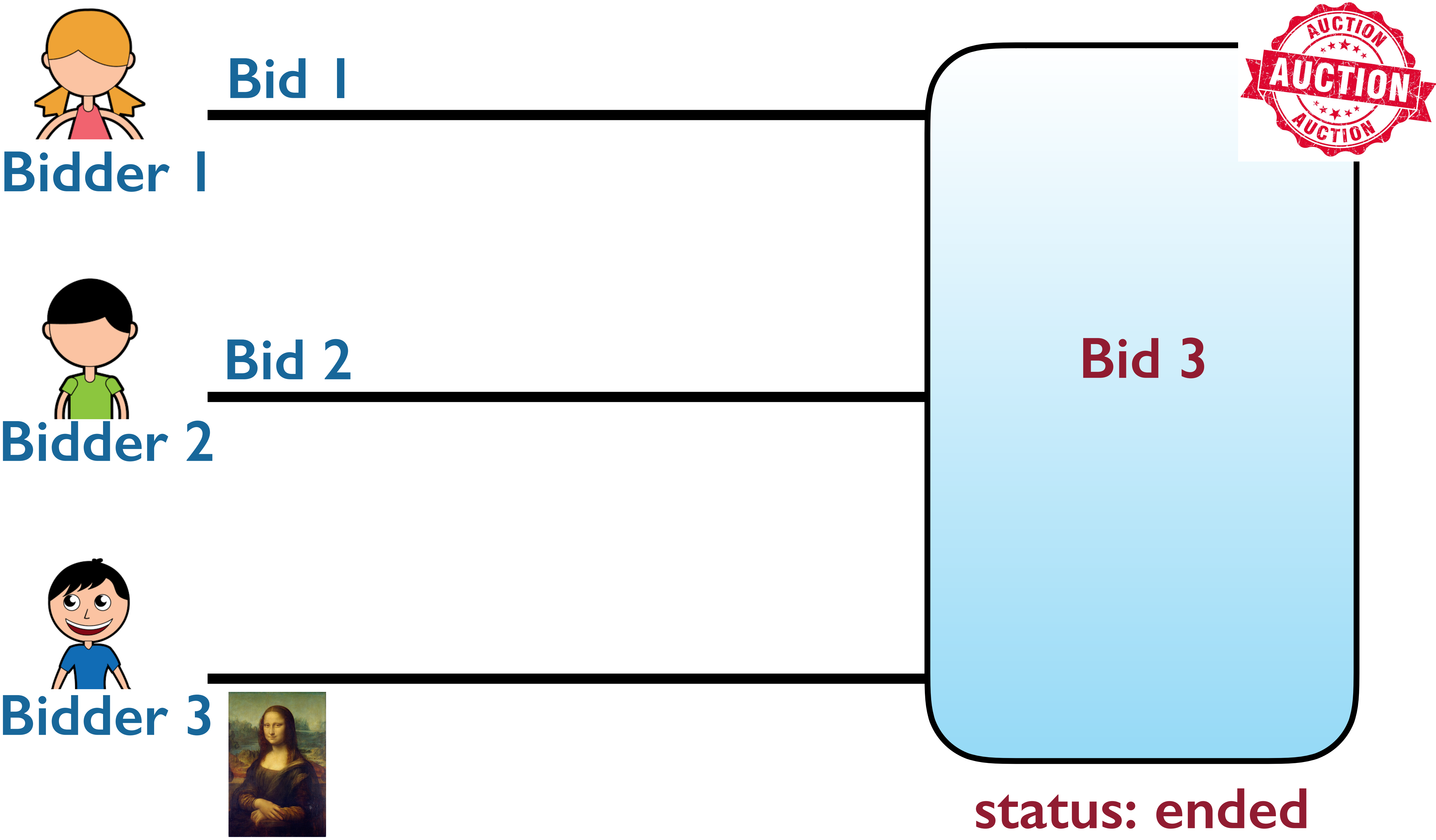
Bid 1

Bid 2

Bid 3

status: ended

Domain Specific Bugs: Auction Contract



Bidder 1

Bid 1

Bidder 2

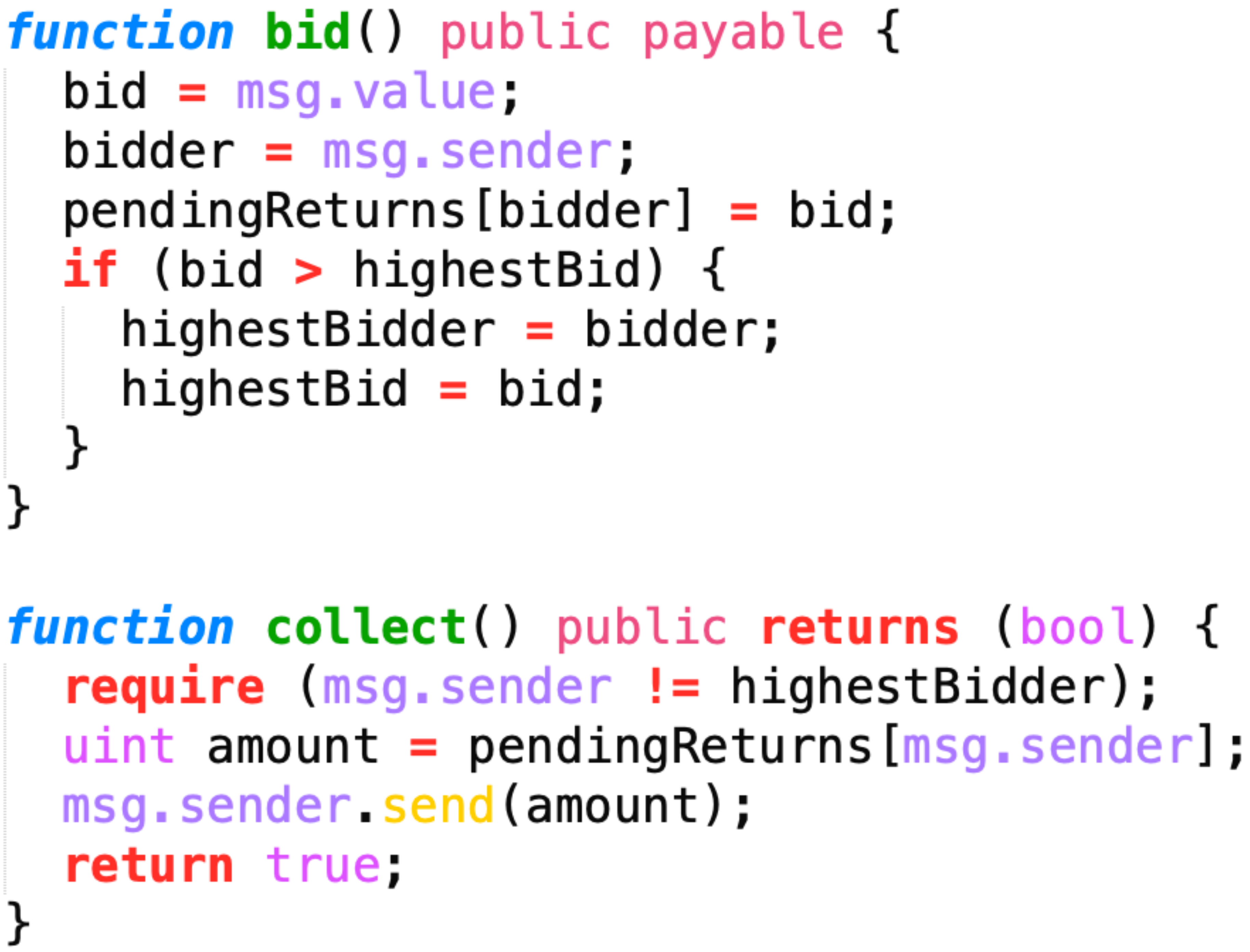
Bid 2

Bidder 3

Bid 3

status: ended

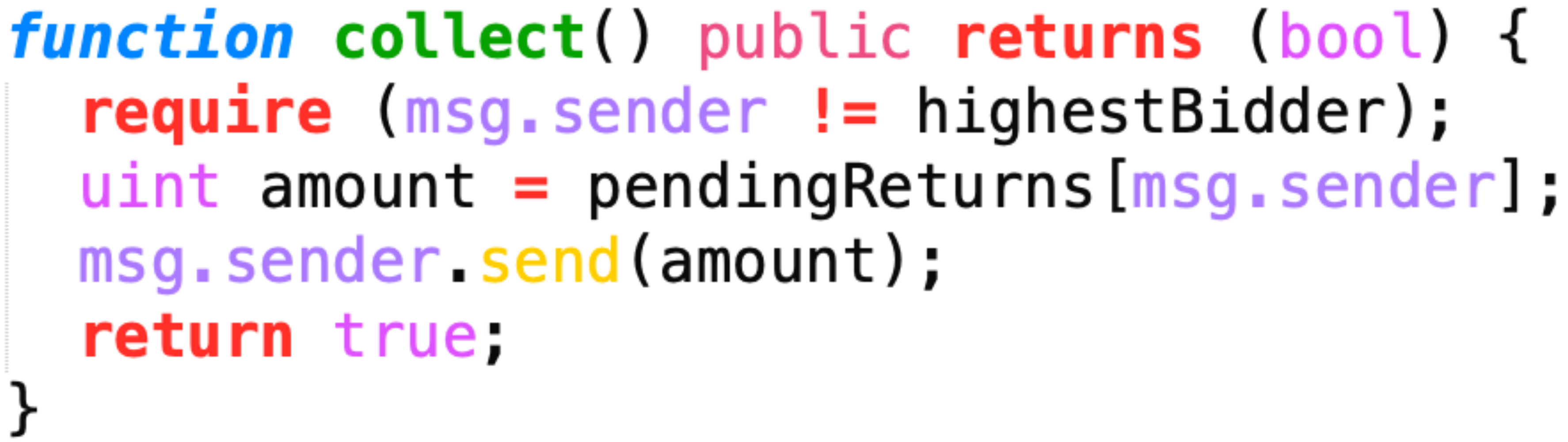
Auction Contract in Solidity



```
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

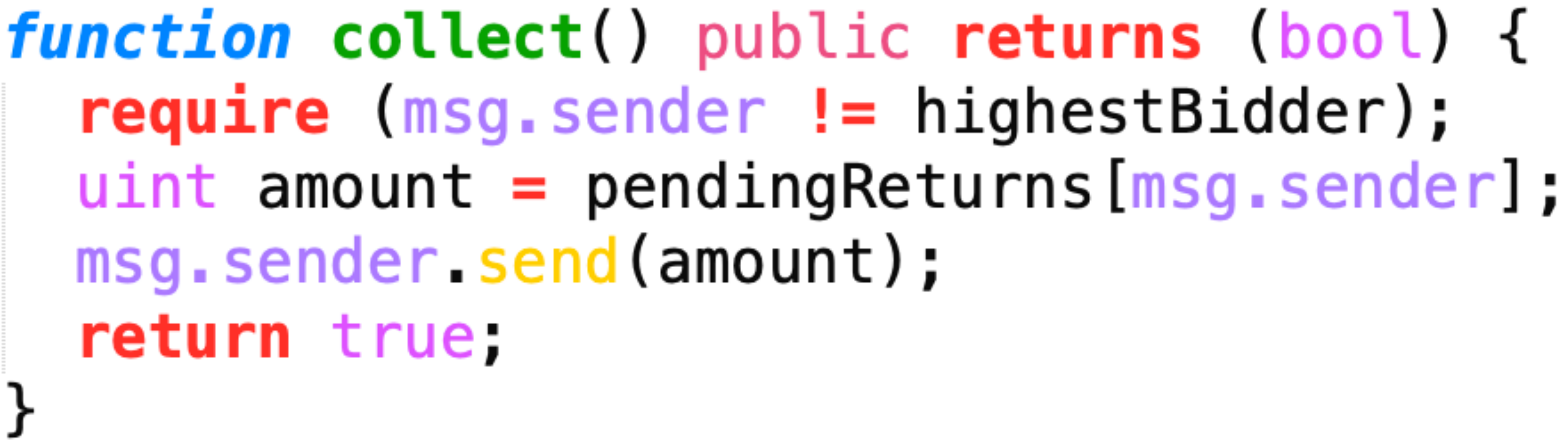


```
function collect() public returns (bool) {
  require (msg.sender != highestBidder);
  uint amount = pendingReturns[msg.sender];
  msg.sender.send(amount);
  return true;
}
```

Auction in Solidity

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

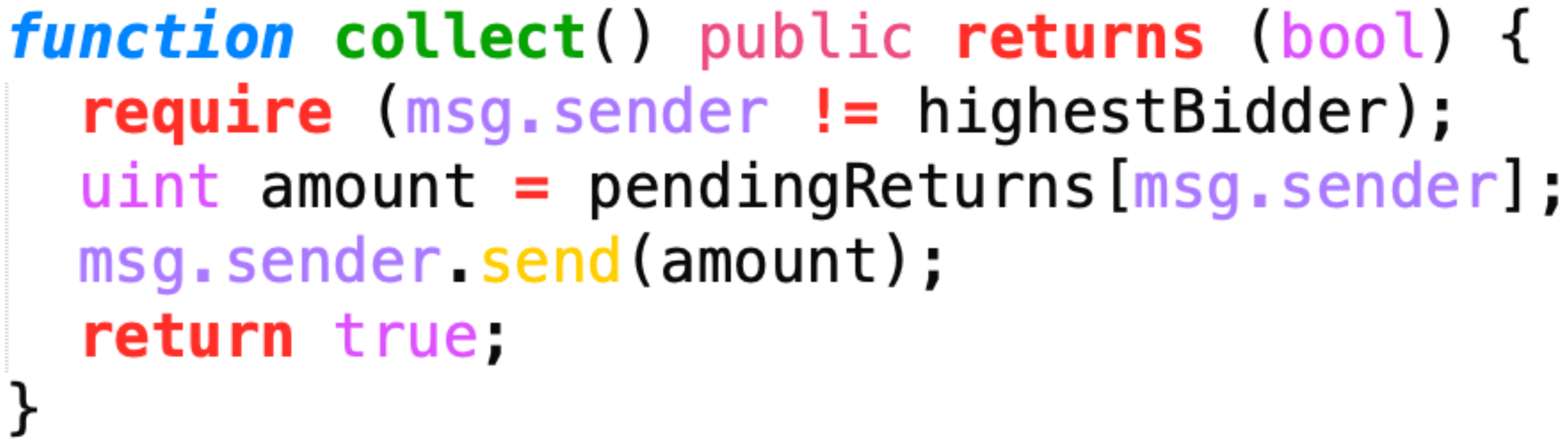
Auction in Solidity



```
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



```
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

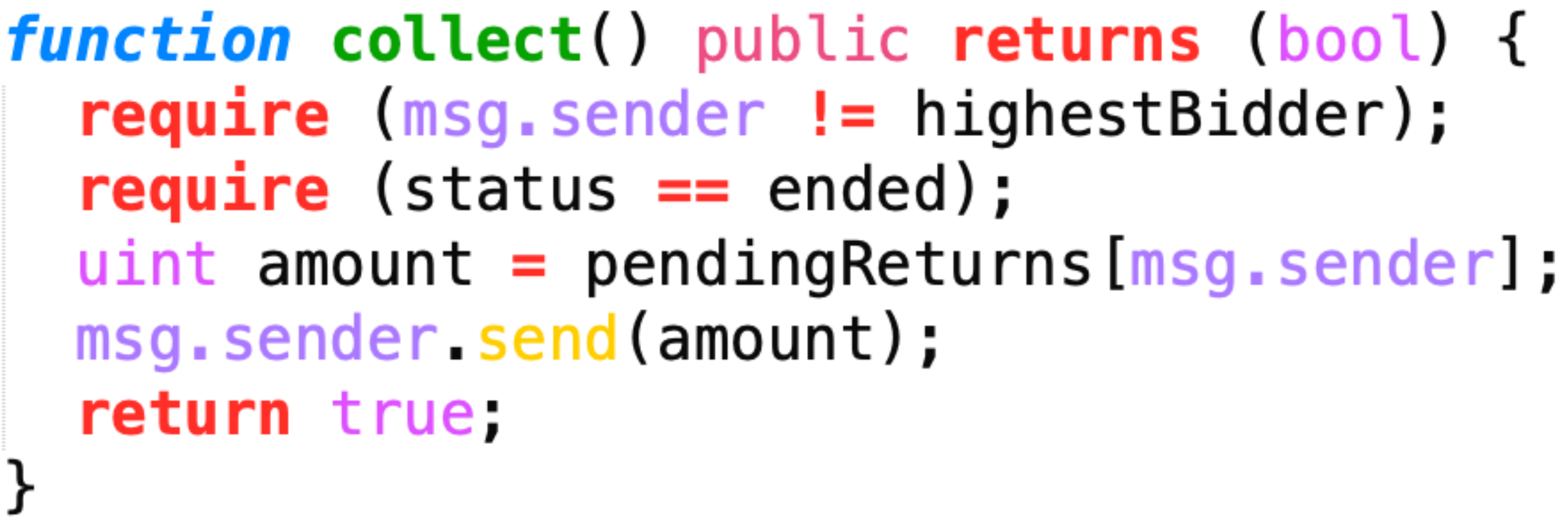
```
function collect() public returns (bool) {
  require (msg.sender != highestBidder);
  uint amount = pendingReturns[msg.sender];
  msg.sender.send(amount);
  return true;
}
```

What happens if collect is called when auction is running?

Protocol is not statically enforced!

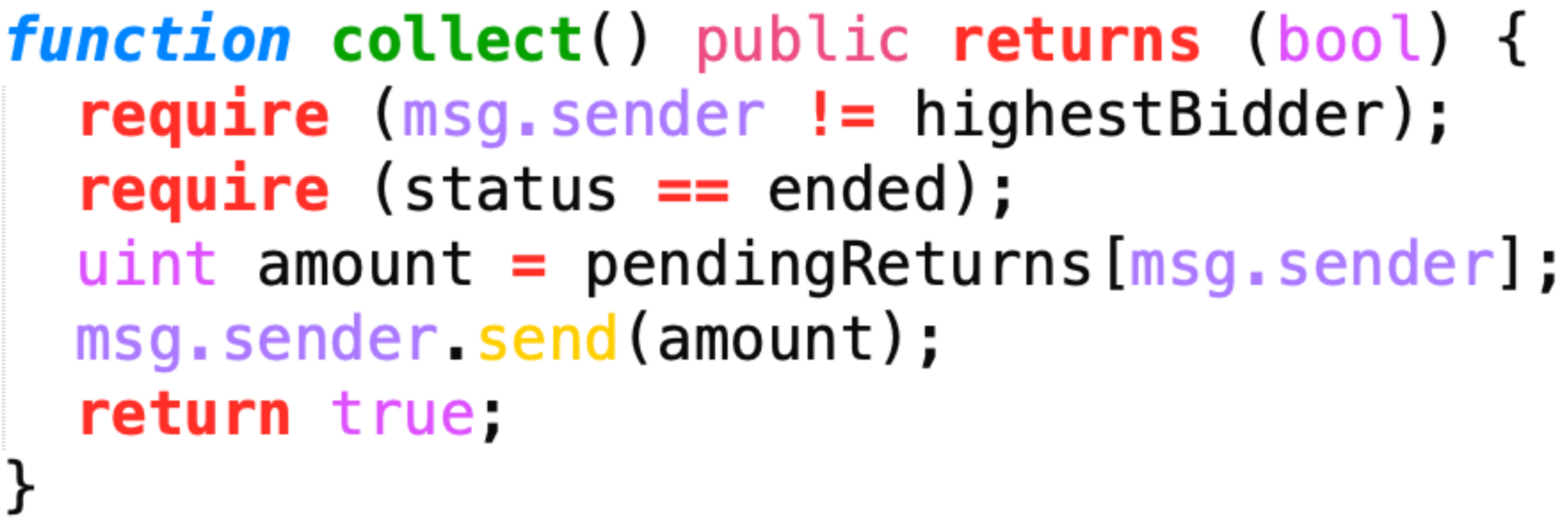
add require (status == ended);

Auction in Solidity



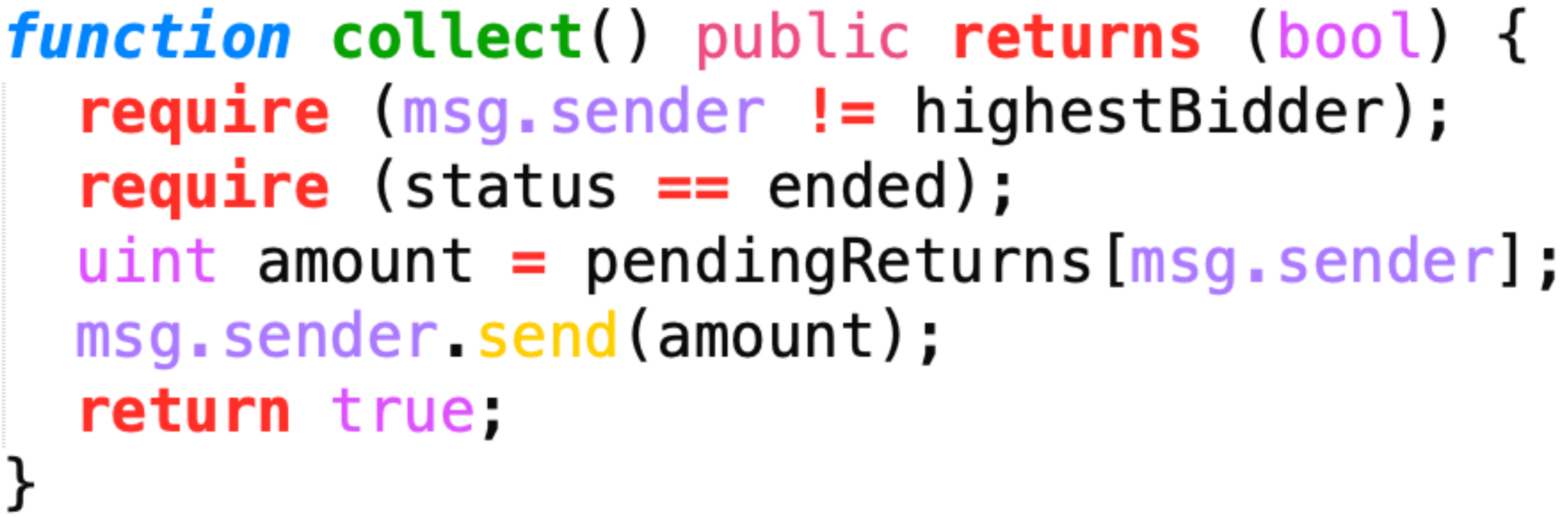
```
function collect() public returns (bool) {
  require (msg.sender != highestBidder);
  require (status == ended);
  uint amount = pendingReturns[msg.sender];
  msg.sender.send(amount);
  return true;
}
```

Auction in Solidity



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

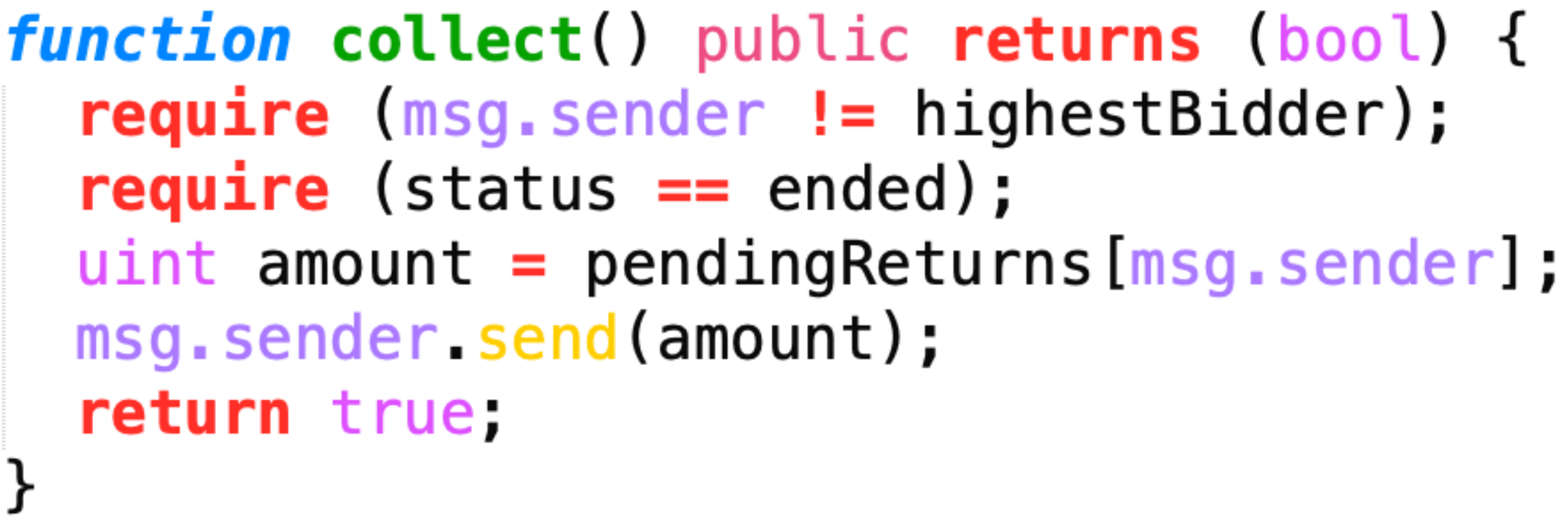
Auction in Solidity



```
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



```
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

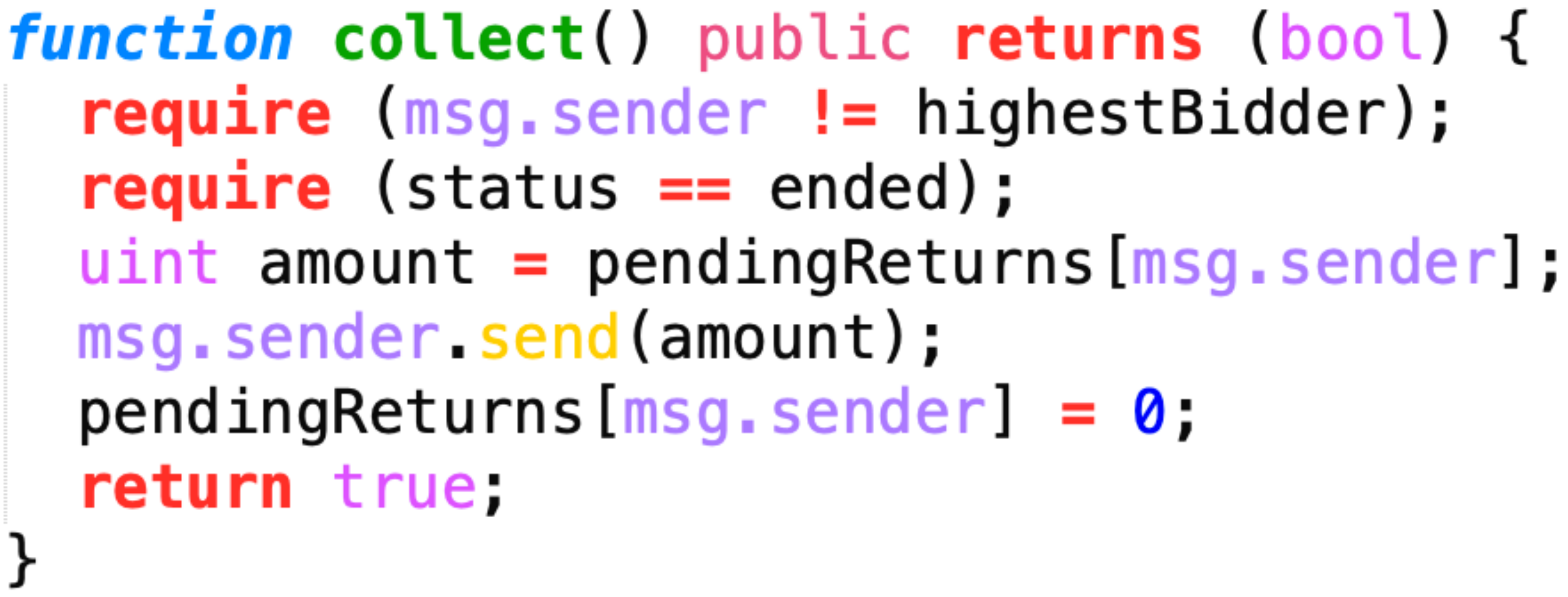
```
function collect() public returns (bool) {
  require (msg.sender != highestBidder);
  require (status == ended);
  uint amount = pendingReturns[msg.sender];
  msg.sender.send(amount);
  return true;
}
```

What happens if collect is called twice?

Linearity is not enforced!

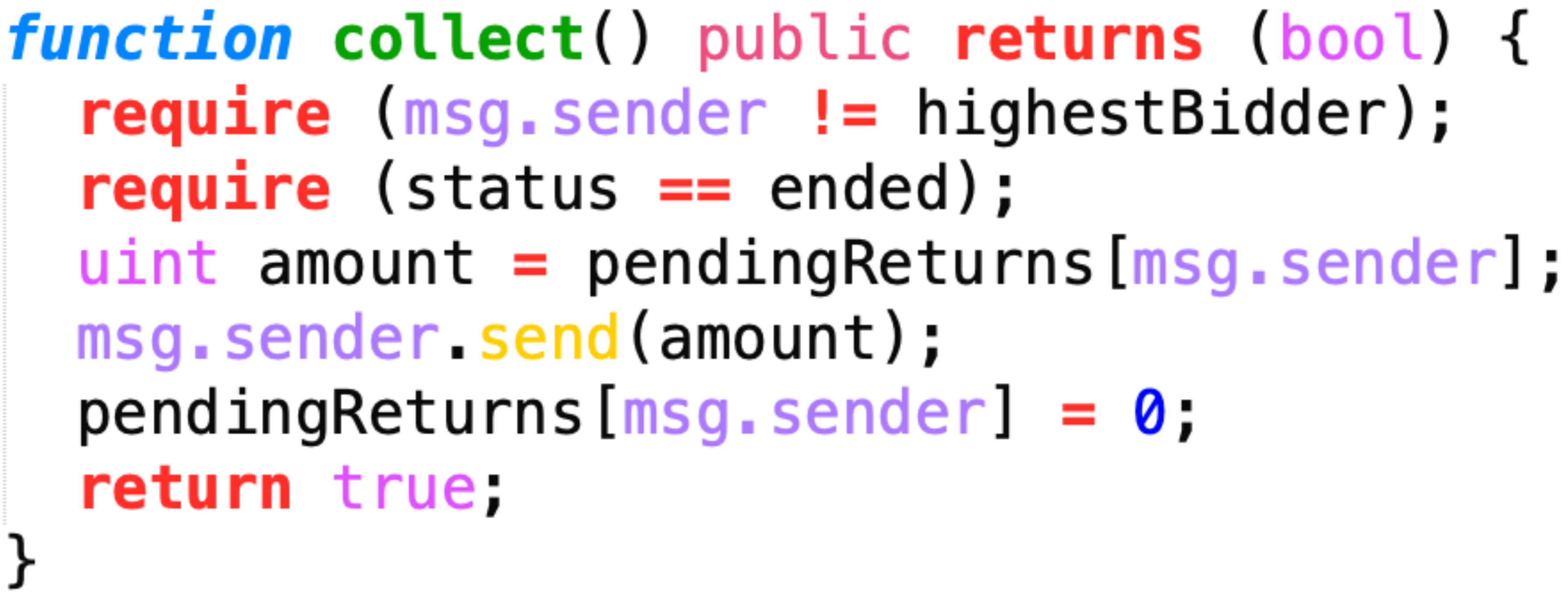
set pendingReturns[msg.sender] = 0

Auction in 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



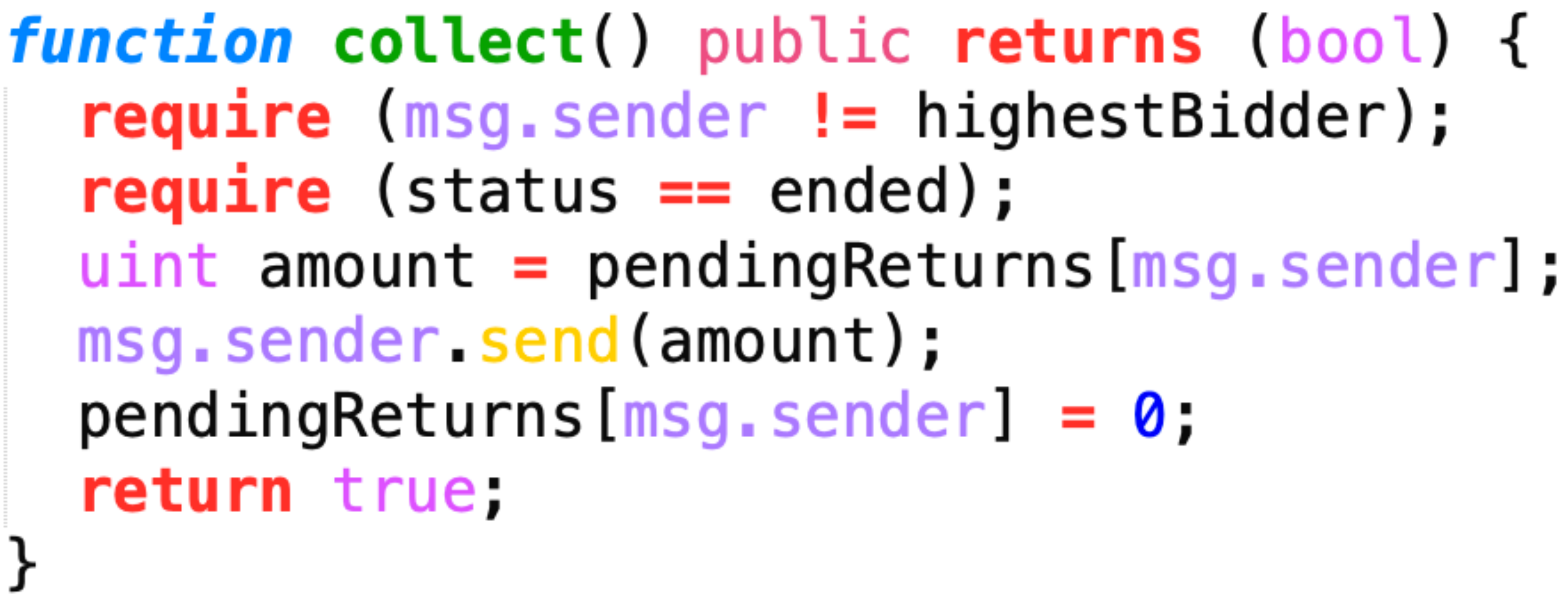
```
function collect() public returns (bool) {
  require (msg.sender != highestBidder);
  require (status == ended);
  uint amount = pendingReturns[msg.sender];
  msg.sender.send(amount);
  pendingReturns[msg.sender] = 0;
  return true;
}
```

Auction in Solidity

```
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



```
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

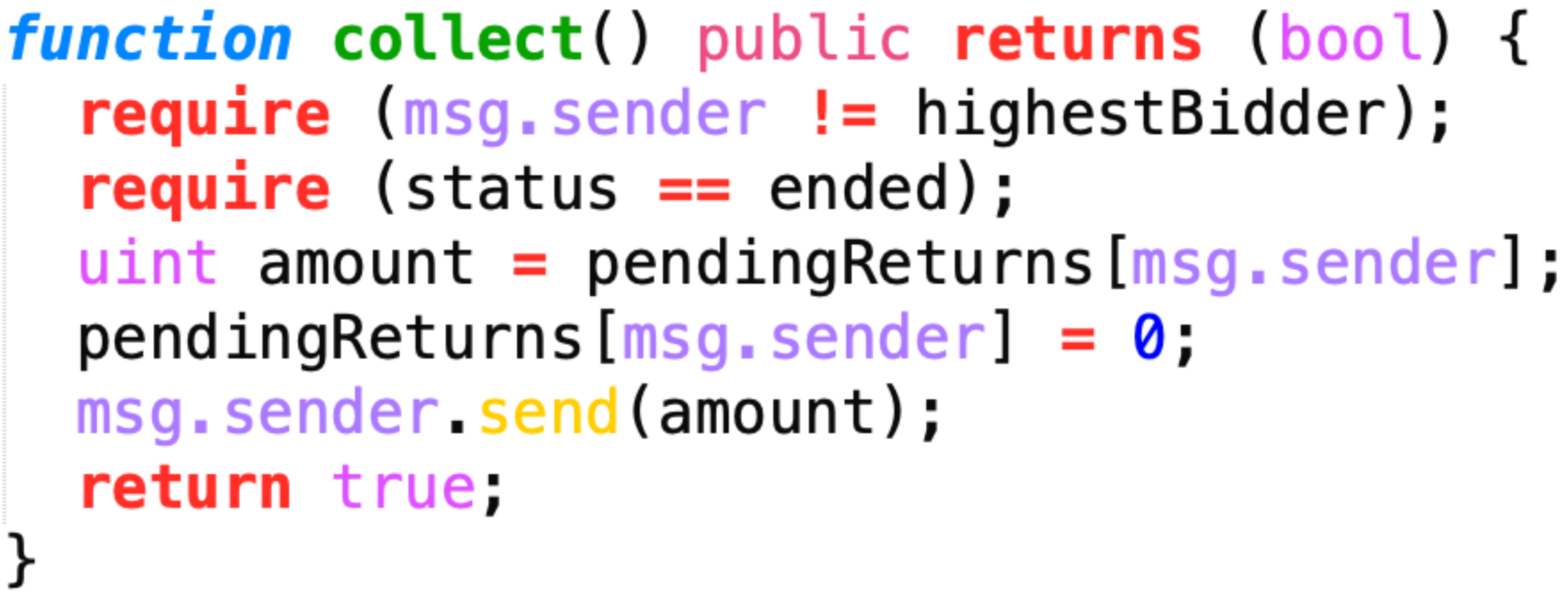
```
function collect() public returns (bool) {
  require (msg.sender != highestBidder);
  require (status == ended);
  uint amount = pendingReturns[msg.sender];
  msg.sender.send(amount);
  pendingReturns[msg.sender] = 0;
  return true;
}
```

Method 'send' potentially transfers control to other contract.

Re-entrancy attack

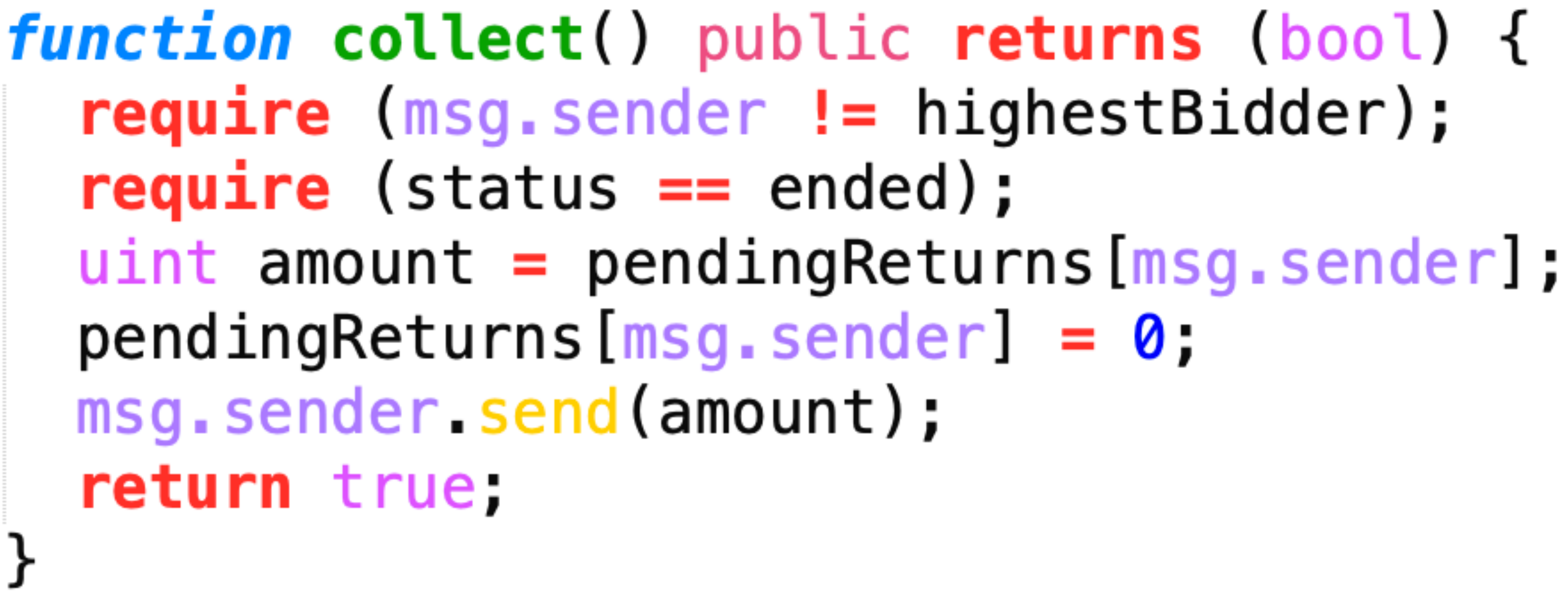
'send' should be the last instruction.

Auction in Solidity



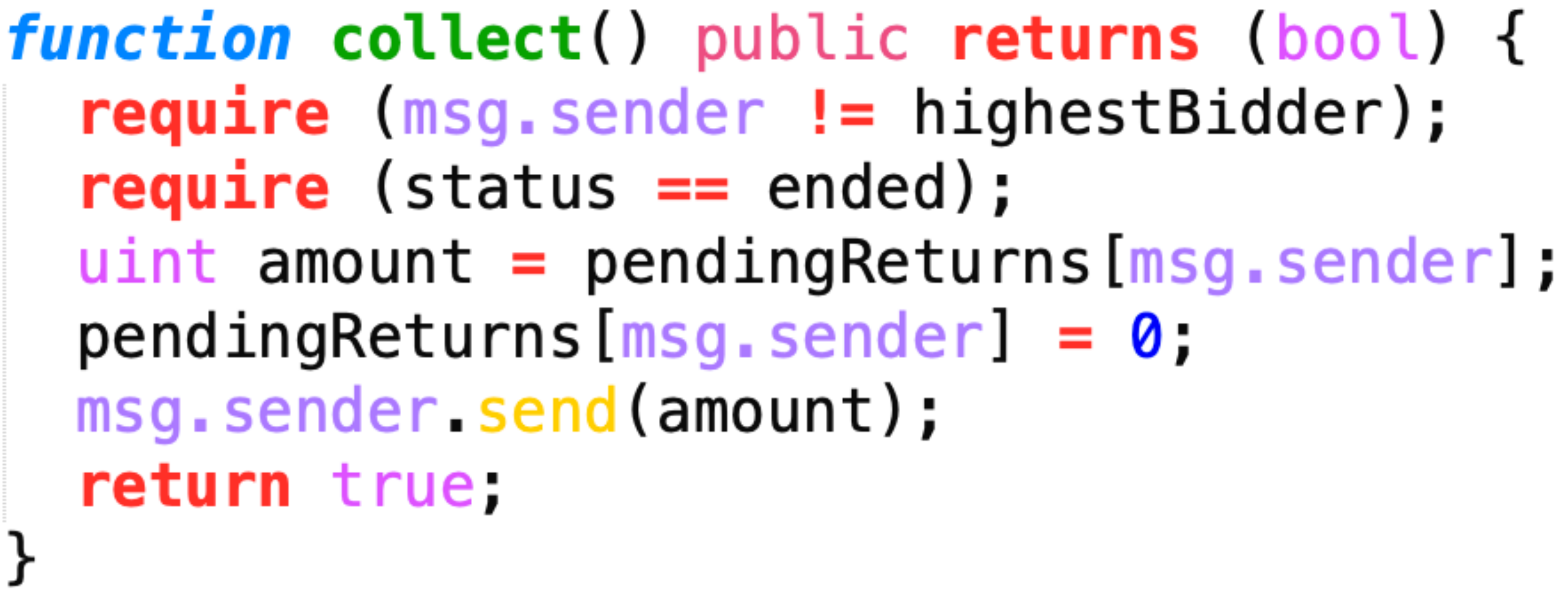
```
function collect() public returns (bool) {
  require (msg.sender != highestBidder);
  require (status == ended);
  uint amount = pendingReturns[msg.sender];
  pendingReturns[msg.sender] = 0;
  msg.sender.send(amount);
  return true;
}
```

Auction in Solidity



```
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;
}
```

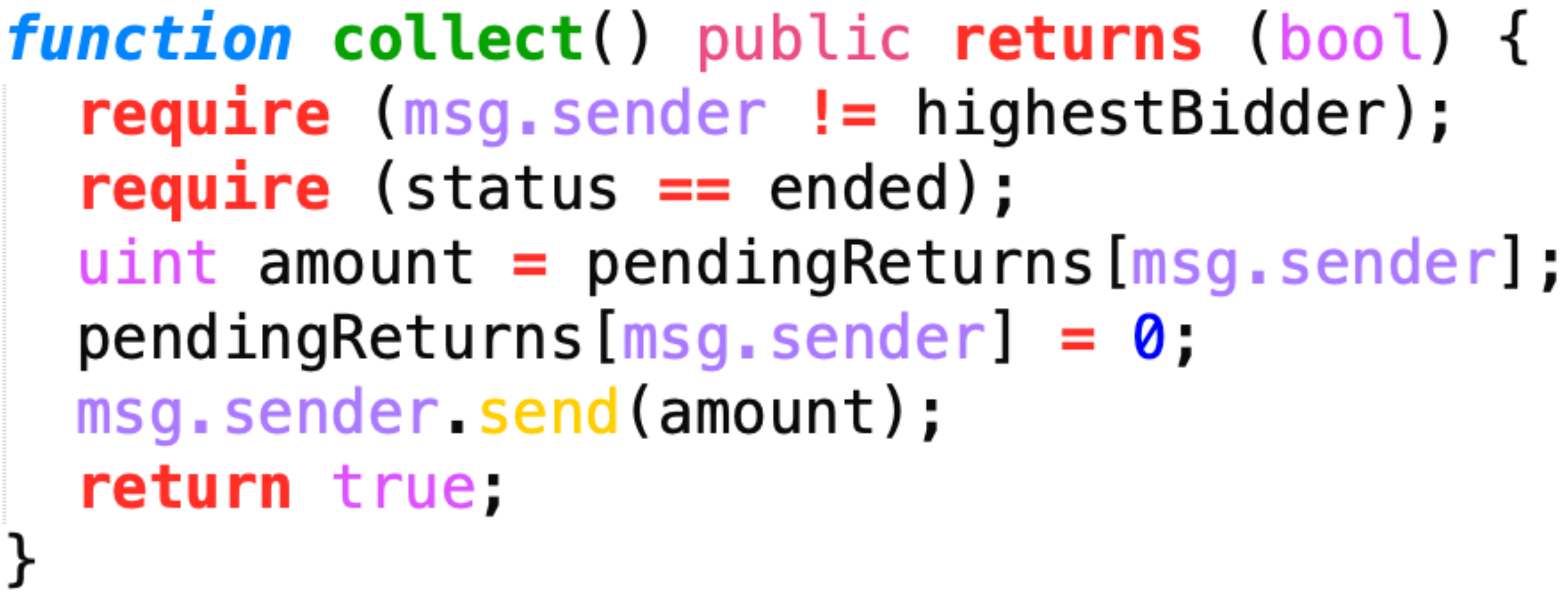
Auction in Solidity



```
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



```
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

```
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

Lead developer:
Ankush Das

A statically-typed, strict, functional language

- Functional fragment of ML

Additional features for domain-specific requirements

	Language feature	Expertise
Gas bounds	Automatic amortized resource analysis	Jan Hoffmann
Tracking assets	Linear type system	Frank Pfenning
Contract interfaces	Shared binary session types	Stephanie Balzer Frank Pfenning

Nomos: A Type-Based Approach

Lead developer:
Ankush Das

A statically-typed, strict, functional language

- Functional fragment of ML

Additional features for domain-specific requirements

	Language feature	Expertise
Gas bounds	Automatic amortized resource analysis	Jan Hoffmann
Tracking assets	Linear type system	Frank Pfenning
Contract interfaces	Shared binary session types	Stephanie Balzer Frank Pfenning

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

Clock cycles, heap space, gas, ...

Given: A (functional) program P

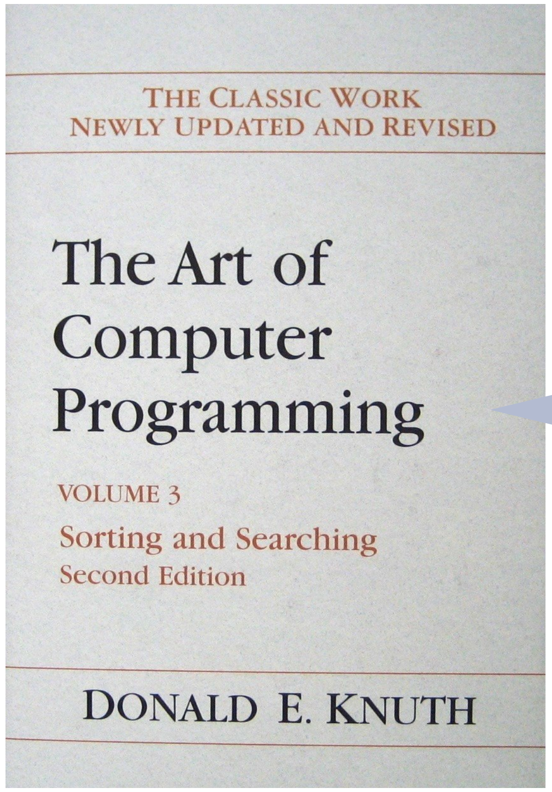
Question: What is the (worst-case) resource consumption of P as a function of the size of its inputs?

Resource Bound Analysis

Given: A (functional) program P

Clock cycles, heap space, gas, ...

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



THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of
Computer
Programming

VOLUME 3
Sorting and Searching
Second Edition

DONALD E. KNUTH

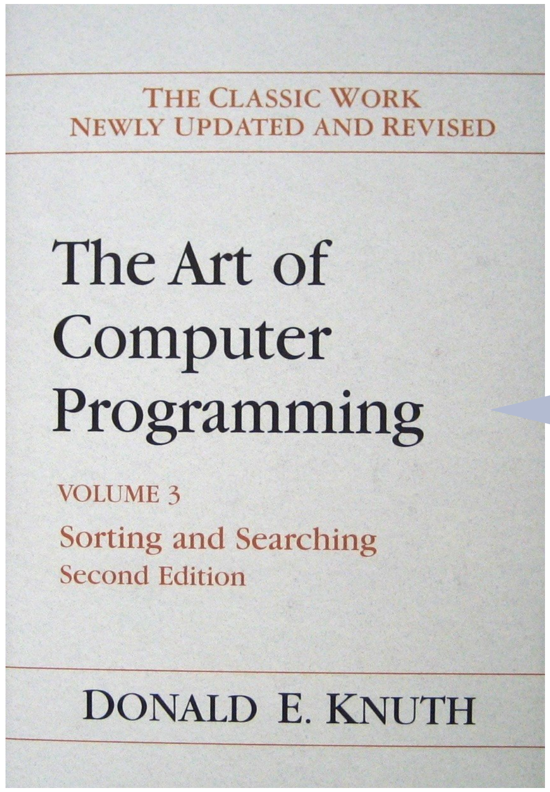
Not only asymptotic bounds but concrete constant factors.

Automatic Resource Bound Analysis

Given: A (functional) program P

Clock cycles, heap space, gas, ...

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



THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of
Computer
Programming

VOLUME 3
Sorting and Searching
Second Edition

DONALD E. KNUTH

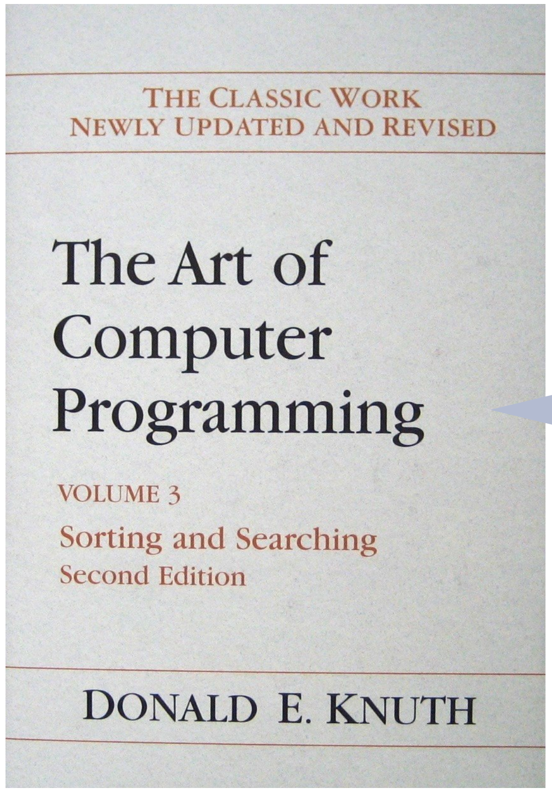
Not only asymptotic bounds but concrete constant factors.

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, ...



THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of
Computer
Programming

VOLUME 3
Sorting and Searching
Second Edition

DONALD E. KNUTH

Not only asymptotic bounds but concrete constant factors.

Goal: produce proofs (easily checkable)

AARA: Use Potential Method

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

$$\Phi(\mathit{state}) \geq 0$$

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

$$\Phi(\mathit{before}) \geq \Phi(\mathit{after}) + \mathit{cost}$$

$\downarrow$ telescoping $\downarrow$

- Initial potential is an upper bound

$$\Phi(\mathit{initial\ state}) \geq \sum \mathit{cost}$$

AARA: Use Potential Method

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

$$\Phi(\mathit{state}) \geq 0$$

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

$$\Phi(\mathit{before}) \geq \Phi(\mathit{after}) + \mathit{cost}$$

↓ telescoping ↓

- Initial potential is an upper bound

$$\Phi(\mathit{initial\ state}) \geq \sum \mathit{cost}$$

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

$$\boxed{\Phi(\mathit{state}) \geq 0}$$

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

$$\boxed{\Phi(\mathit{before}) \geq \Phi(\mathit{after}) + \mathit{cost}}$$

↓ telescoping ↓

- Initial potential is an upper bound

$$\boxed{\Phi(\mathit{initial\ state}) \geq \sum \mathit{cost}}$$

Type systems for automatic analysis

Clear soundness theorem.
Compositional. Efficient inference.

- 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

```
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:

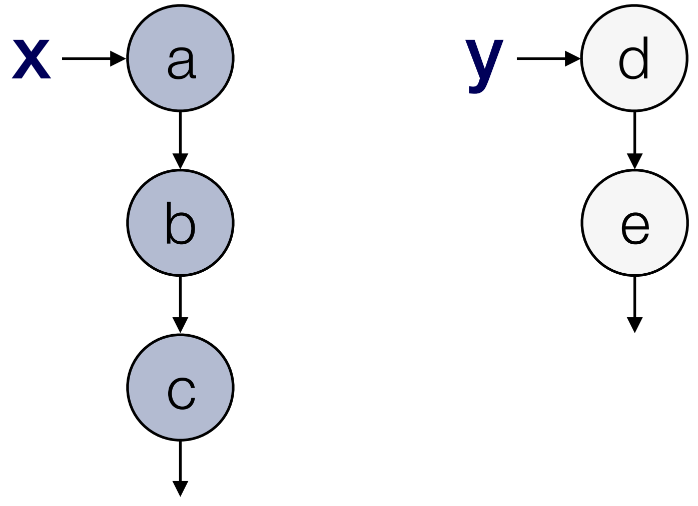
Example: Append for Persistent Lists

```
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:



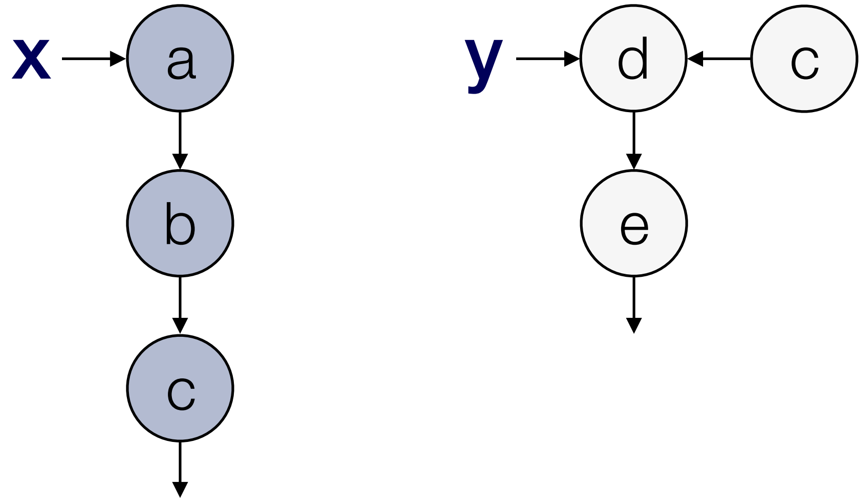
Example: Append for Persistent Lists

```
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:



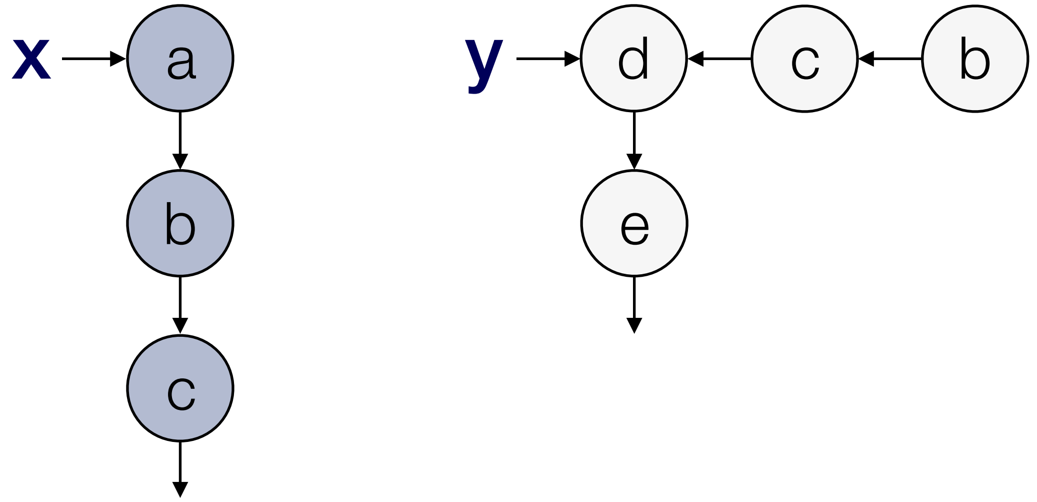
Example: Append for Persistent Lists

```
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:



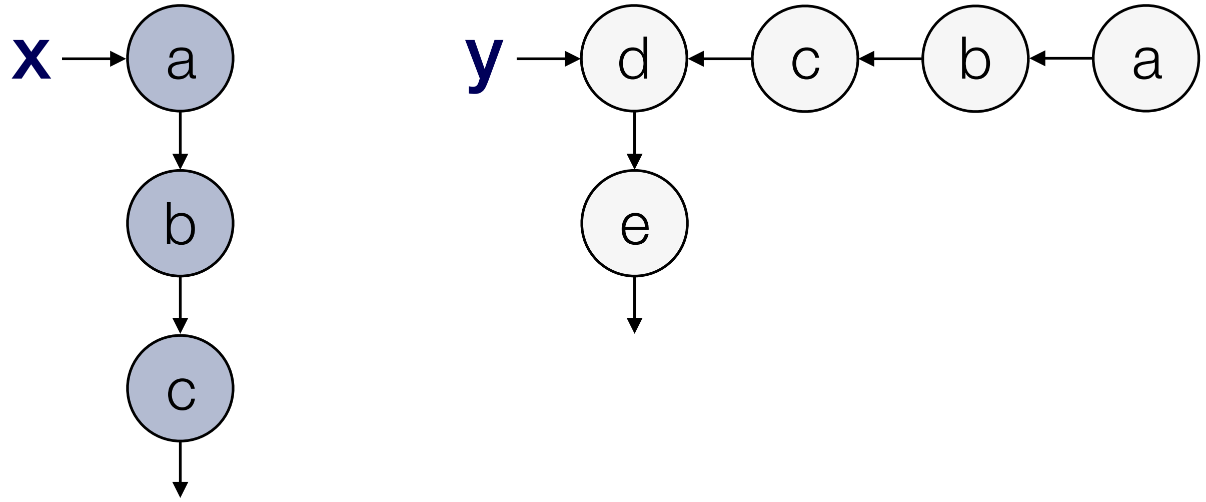
Example: Append for Persistent Lists

```
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:



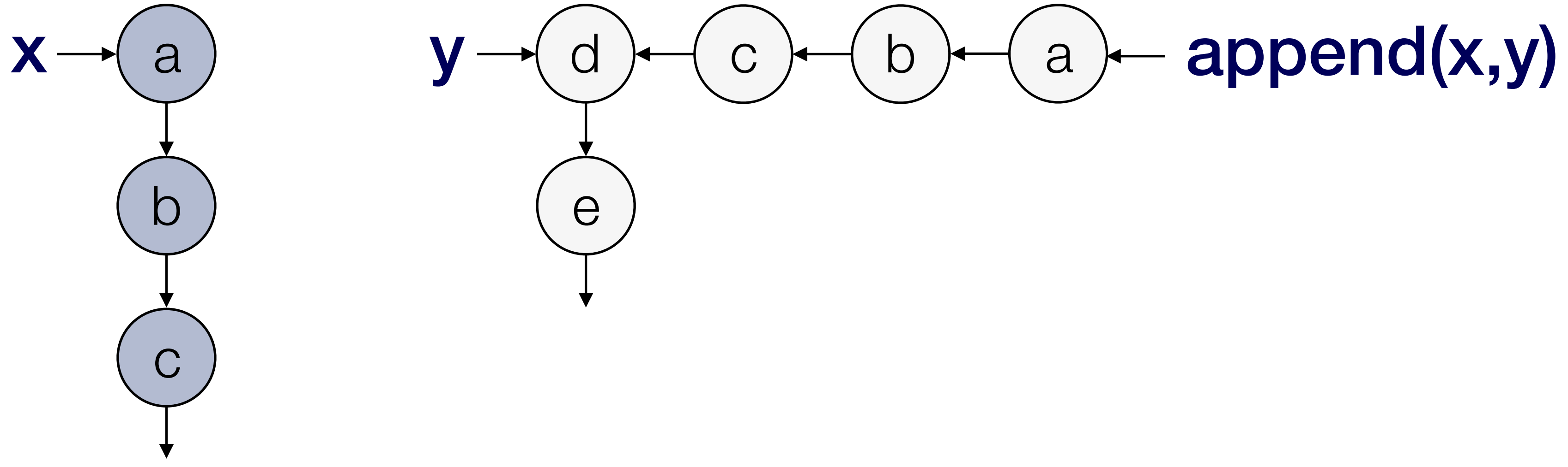
Example: Append for Persistent Lists

```
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:



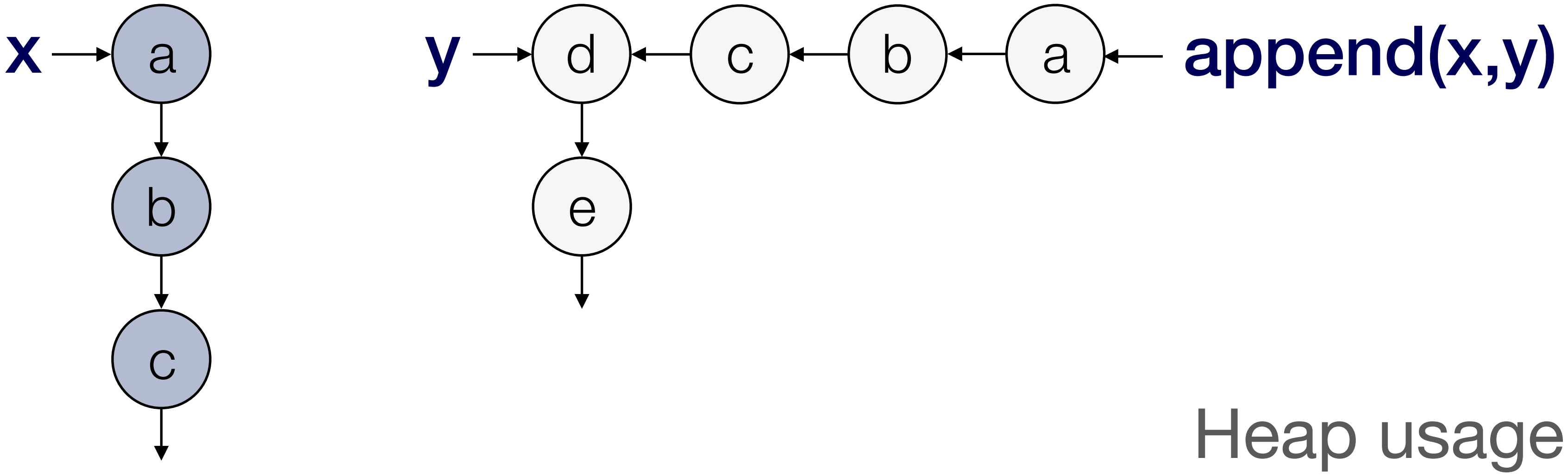
Example: Append for Persistent Lists

```
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:



Heap usage: 2*n = 2*3 = 6

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

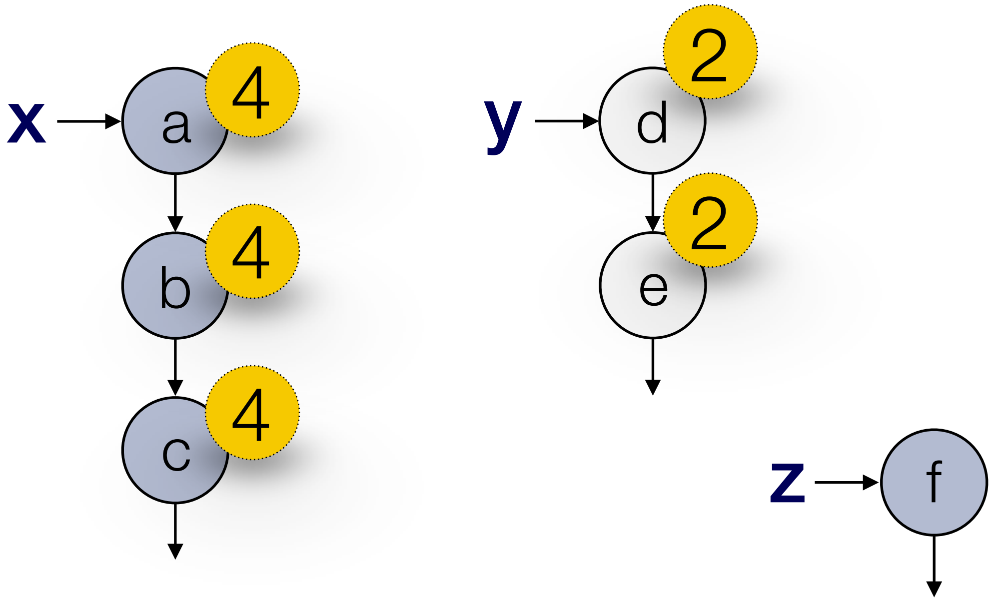
- n is the length of list x
- m is the length of list y

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



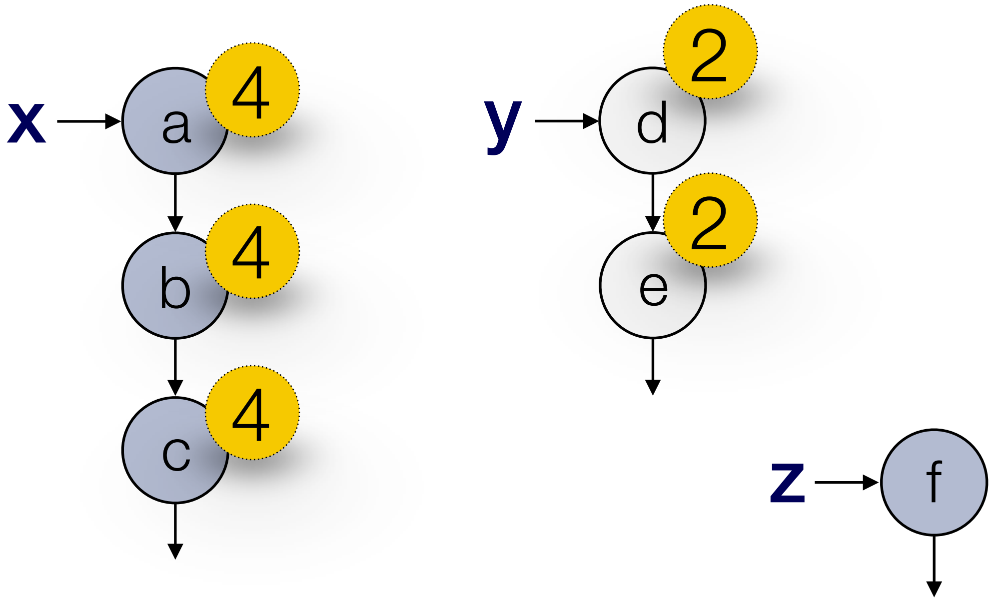
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

Example: Composing Calls of Append

```
f(x,y,z) =
 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



append(x,y)

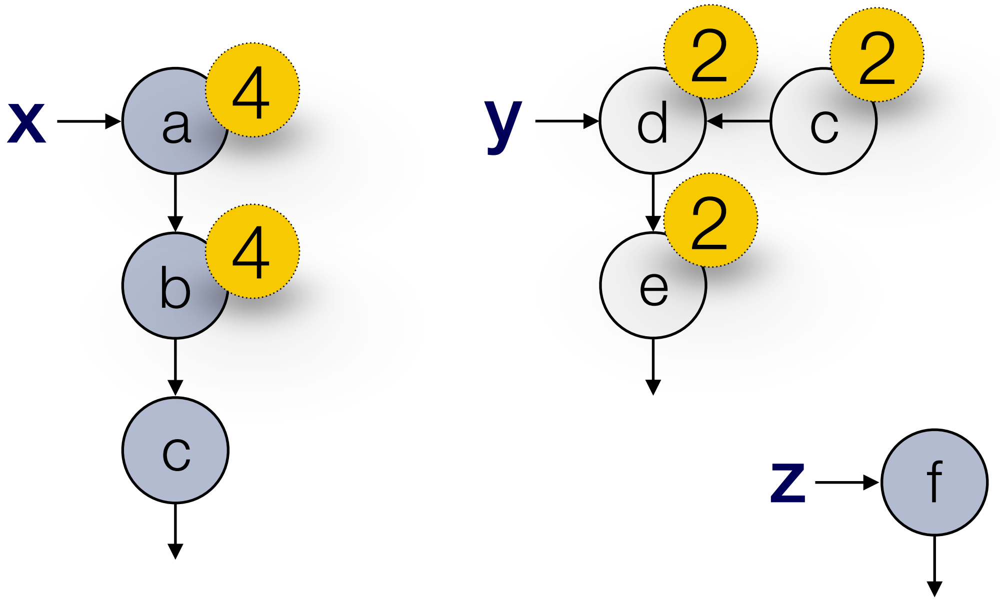
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

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



append(x,y)

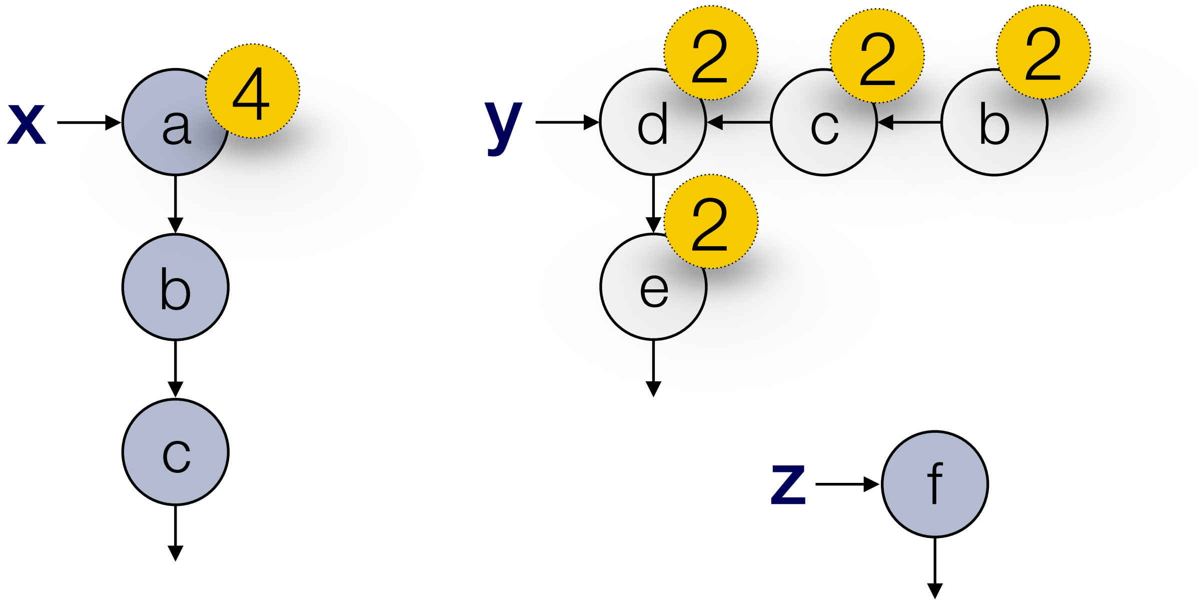
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

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



append(x,y)

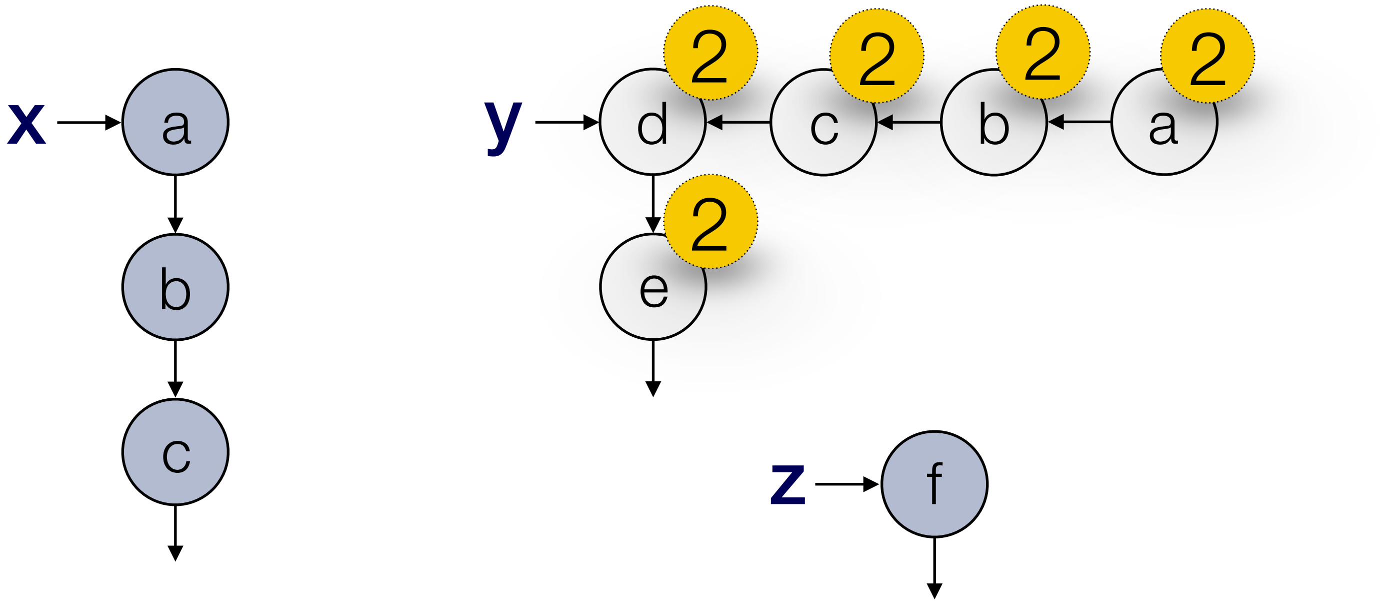
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

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



append(x,y)

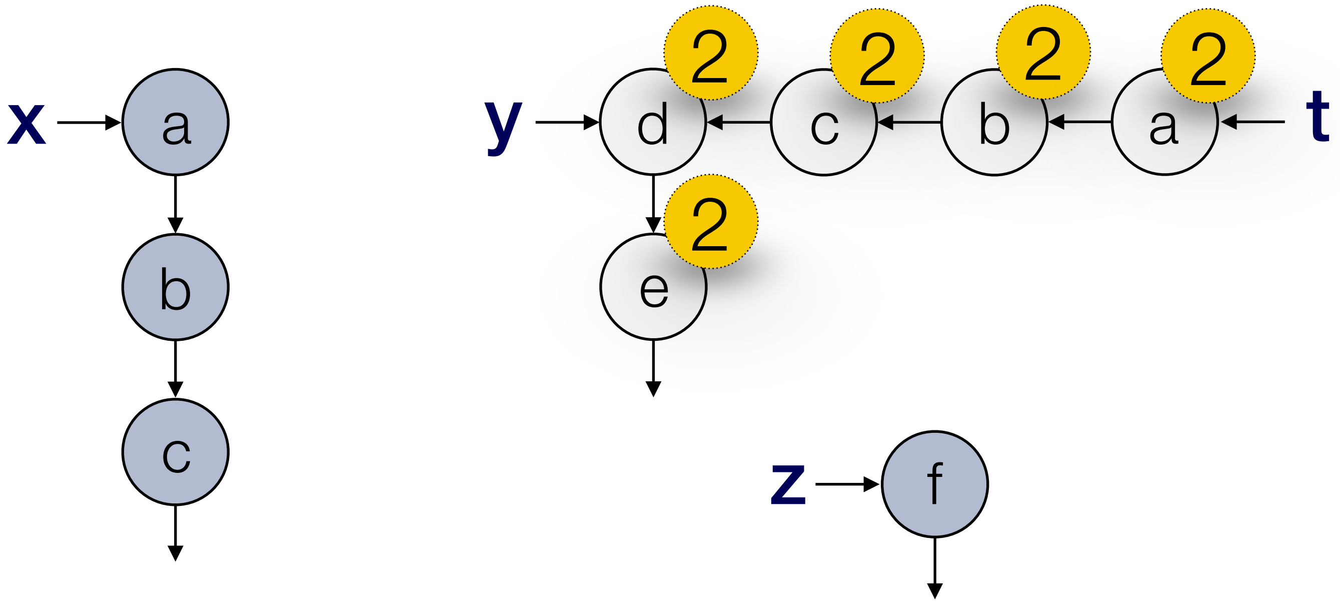
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

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



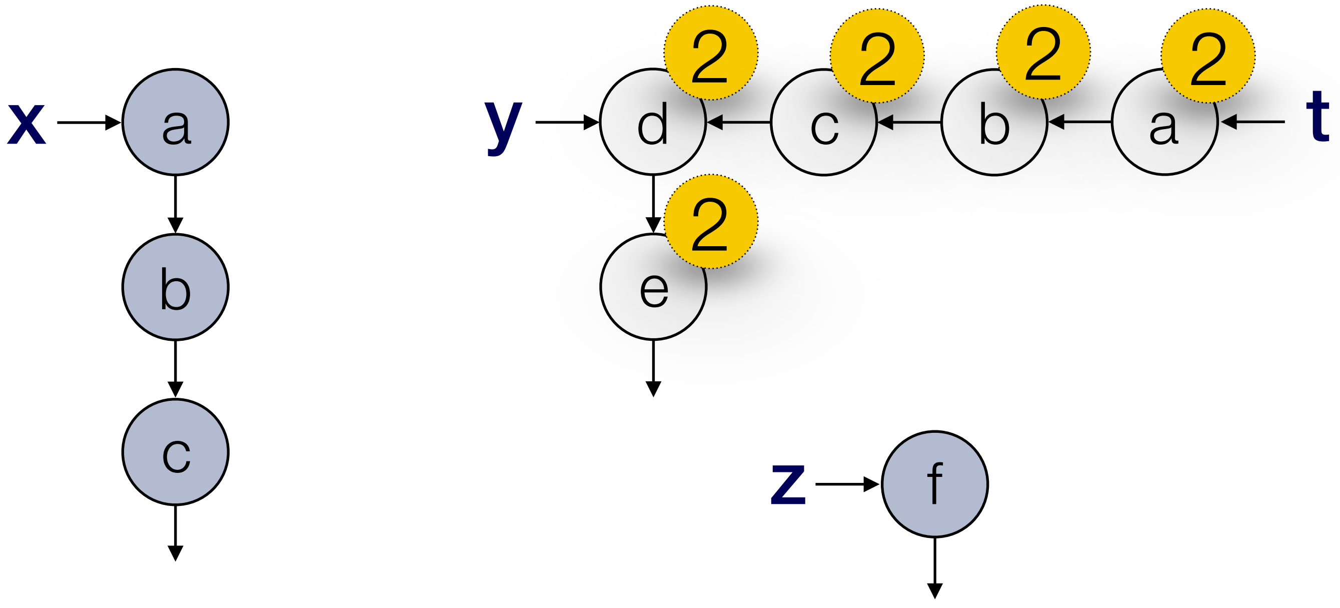
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

Example: Composing Calls of Append

```
f(x,y,z) =
  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



append(t,z)

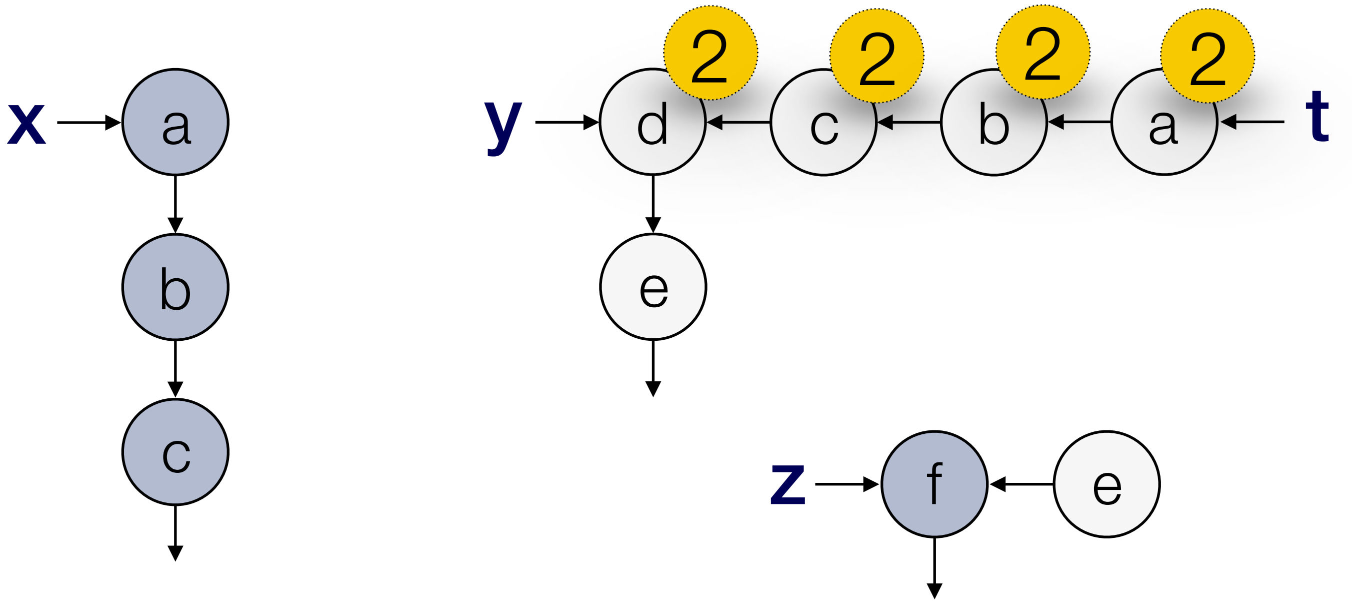
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

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



append(t,z)

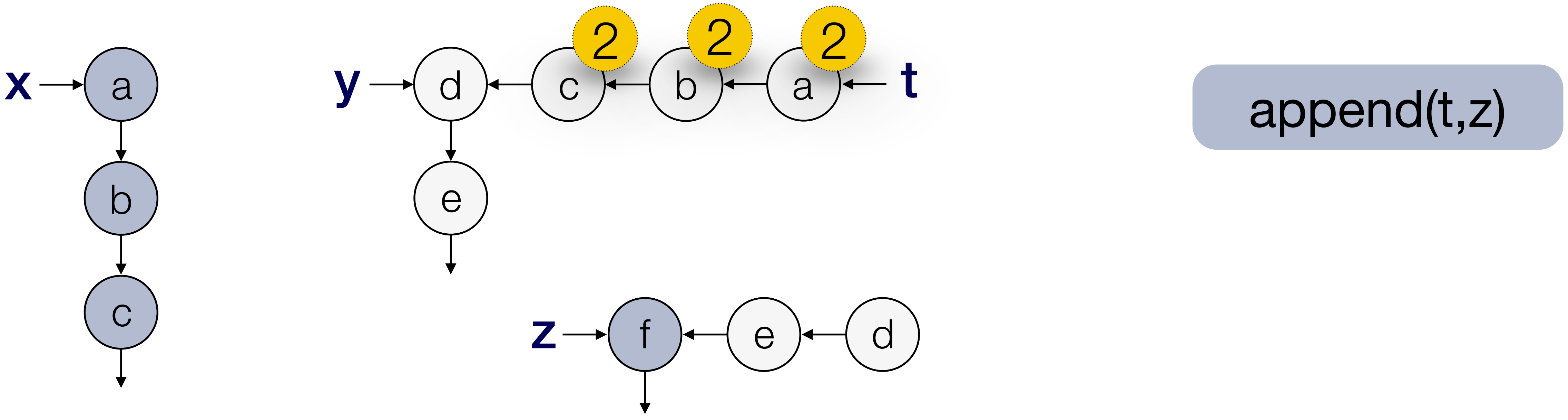
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

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



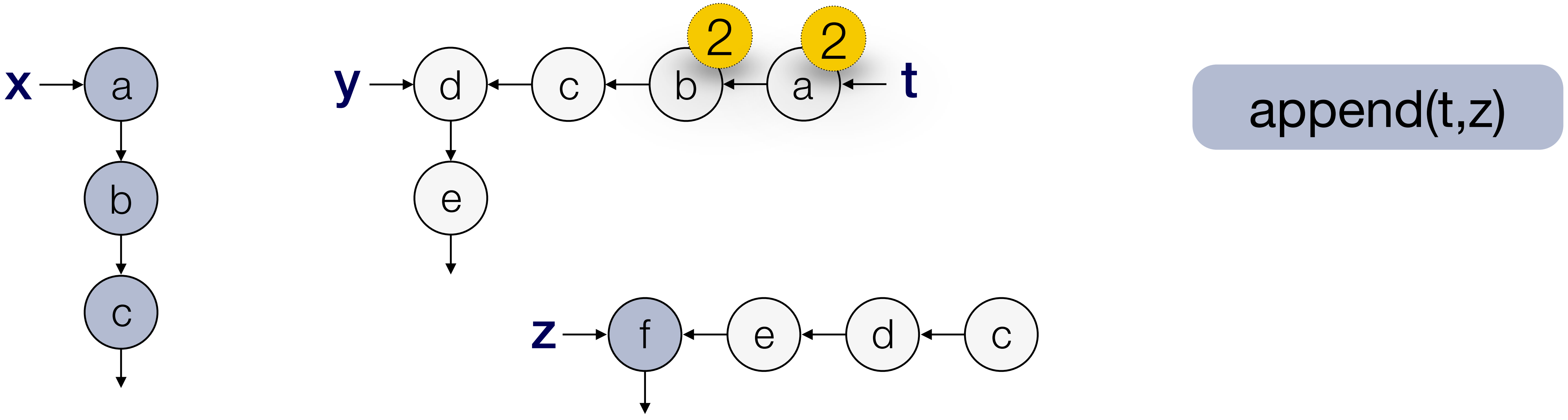
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

Example: Composing Calls of Append

```
f(x,y,z) =
 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



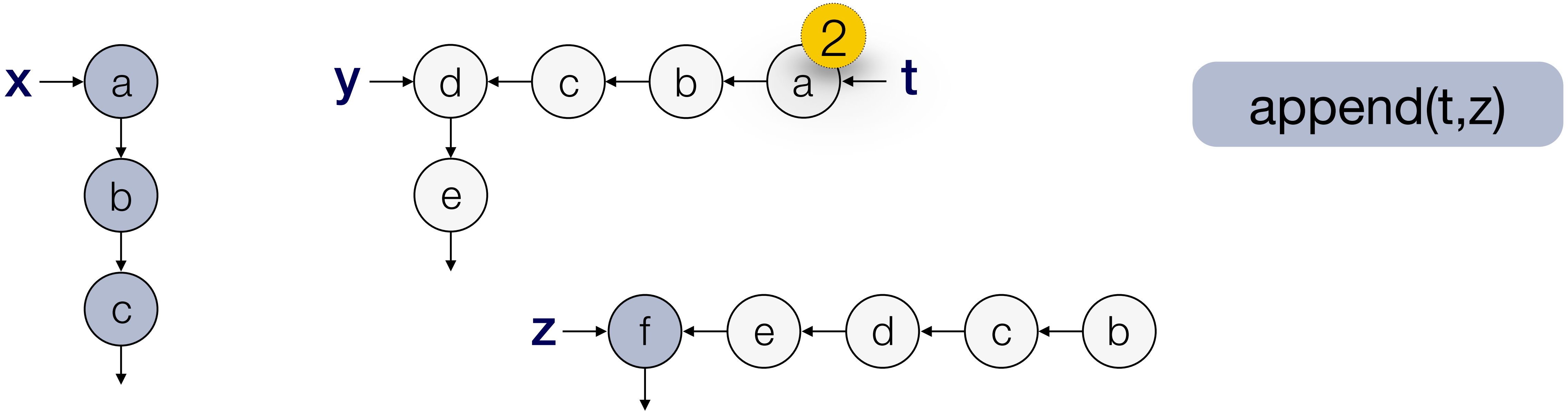
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

Example: Composing Calls of Append

```
f(x,y,z) =
 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



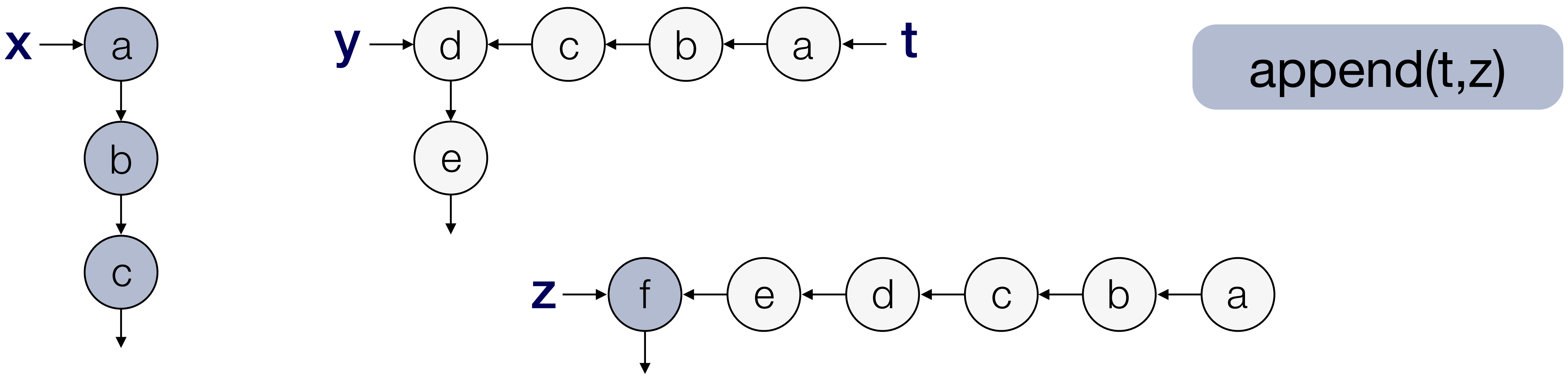
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

Example: Composing Calls of Append

```
f(x,y,z) =
 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



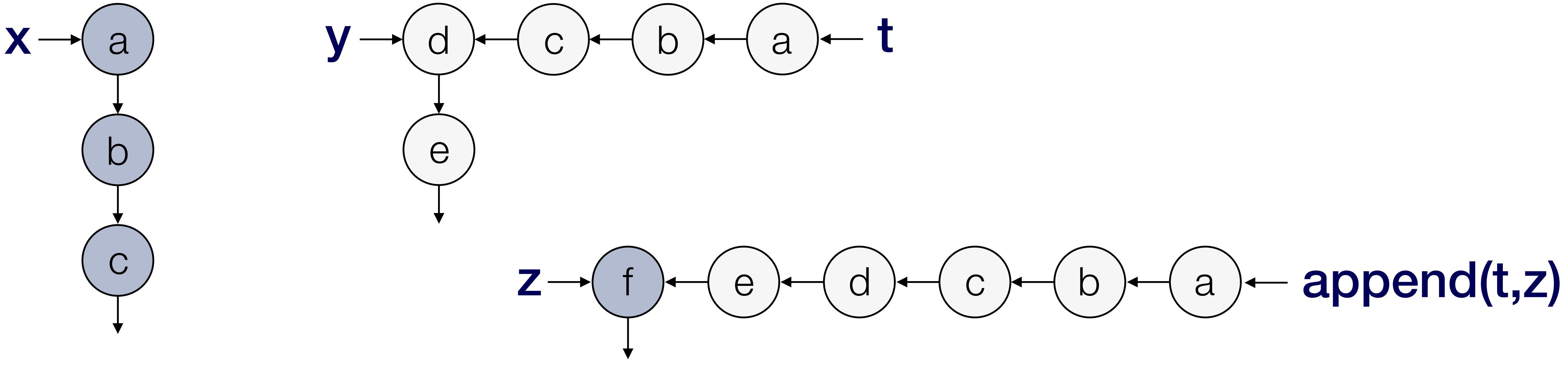
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

Example: Composing Calls of Append

```
f(x,y,z) =
 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



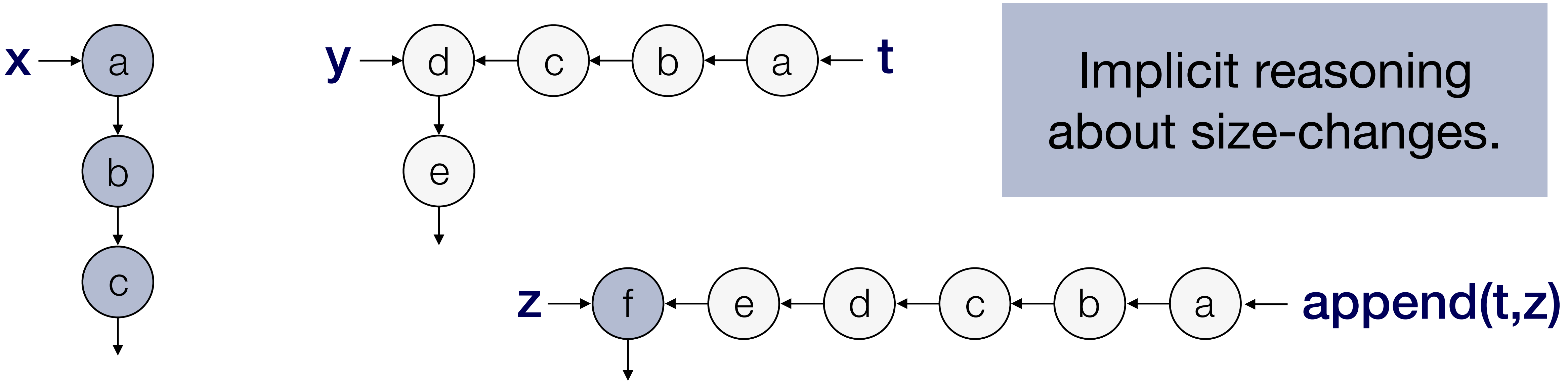
Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

Example: Composing Calls of Append

```
f(x,y,z) =
 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



Implicit reasoning about size-changes.

Initial potential: 4*n + 2*m = 4*3 + 2*2 = 16

Example: Composing Calls of Append

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

$$\texttt{append: }(L^{4}(\texttt{int}),L^{2}(\texttt{int}))\xrightarrow{0/0}L^{2}(\texttt{int})$$

$$\texttt{append: }(L^{2}(\texttt{int}),L^{0}(\texttt{int}))\xrightarrow{0/0}L^{0}(\texttt{int})$$

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

$$\texttt{append: }(L^{q}(\texttt{int}),L^{p}(\texttt{int}))\xrightarrow{s/t}L^{r}(\texttt{int})\mid\Phi$$

Linear constraints.

	Linear 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	✓

Polynomial Potential Functions

	Linear 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

	Linear Potential Functions	Multivariate 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

For example $m*n^2$.

	Linear Potential Functions	Multivariate 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	✓	✓

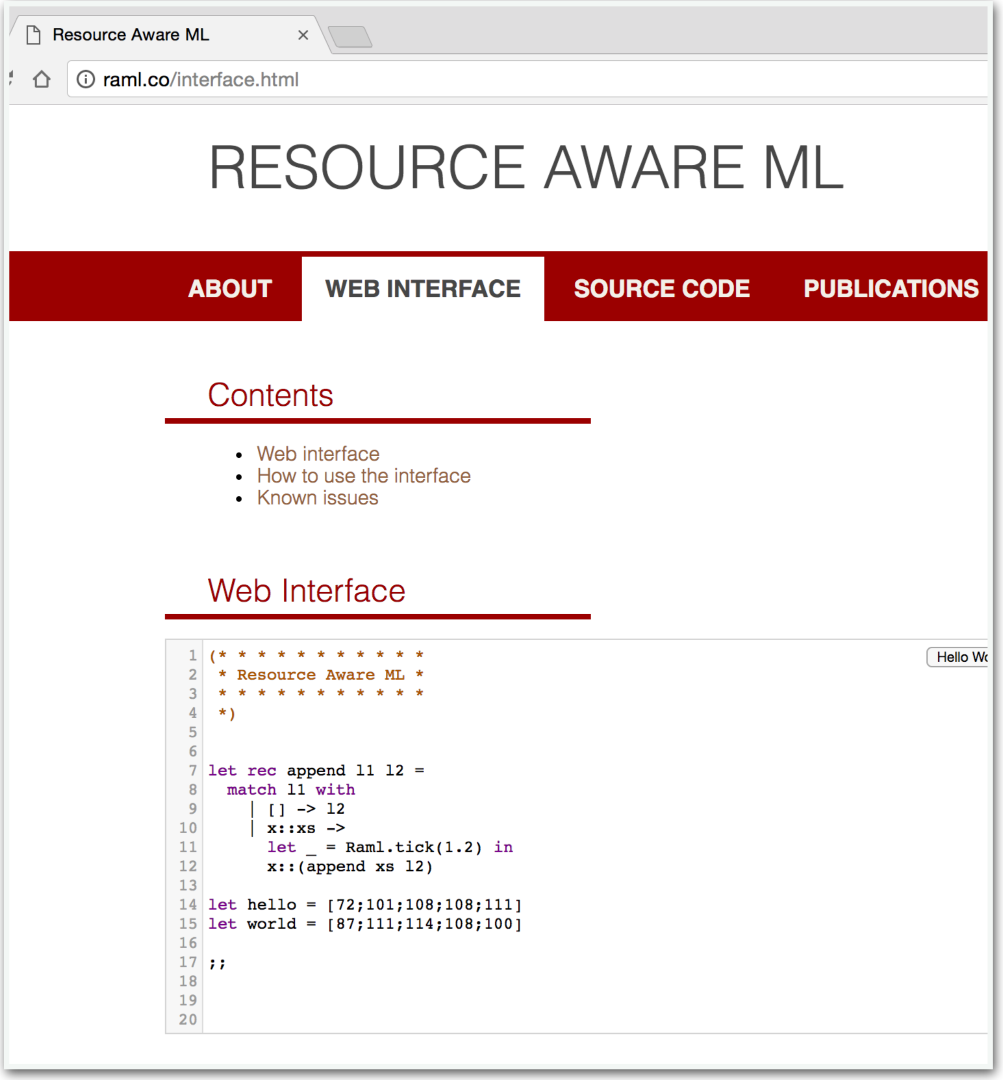
Strong soundness theorem.

Polynomial Potential Functions

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)



RESOURCE AWARE ML

Contents

- Web interface
- How to use the interface
- Known issues

Web Interface

```
(* * * * * * * * * * * *
 * Resource Aware ML *
 * * * * * * * * * * *
 *)


let rec append l1 l2 =
  match l1 with
    | [] -> l2
    | x::xs ->
      let _ = Raml.tick(1.2) in
      x::(append xs l2)

let hello = [72;101;108;108;111]
let world = [87;111;114;108;100]

;;
```

http://raml.co

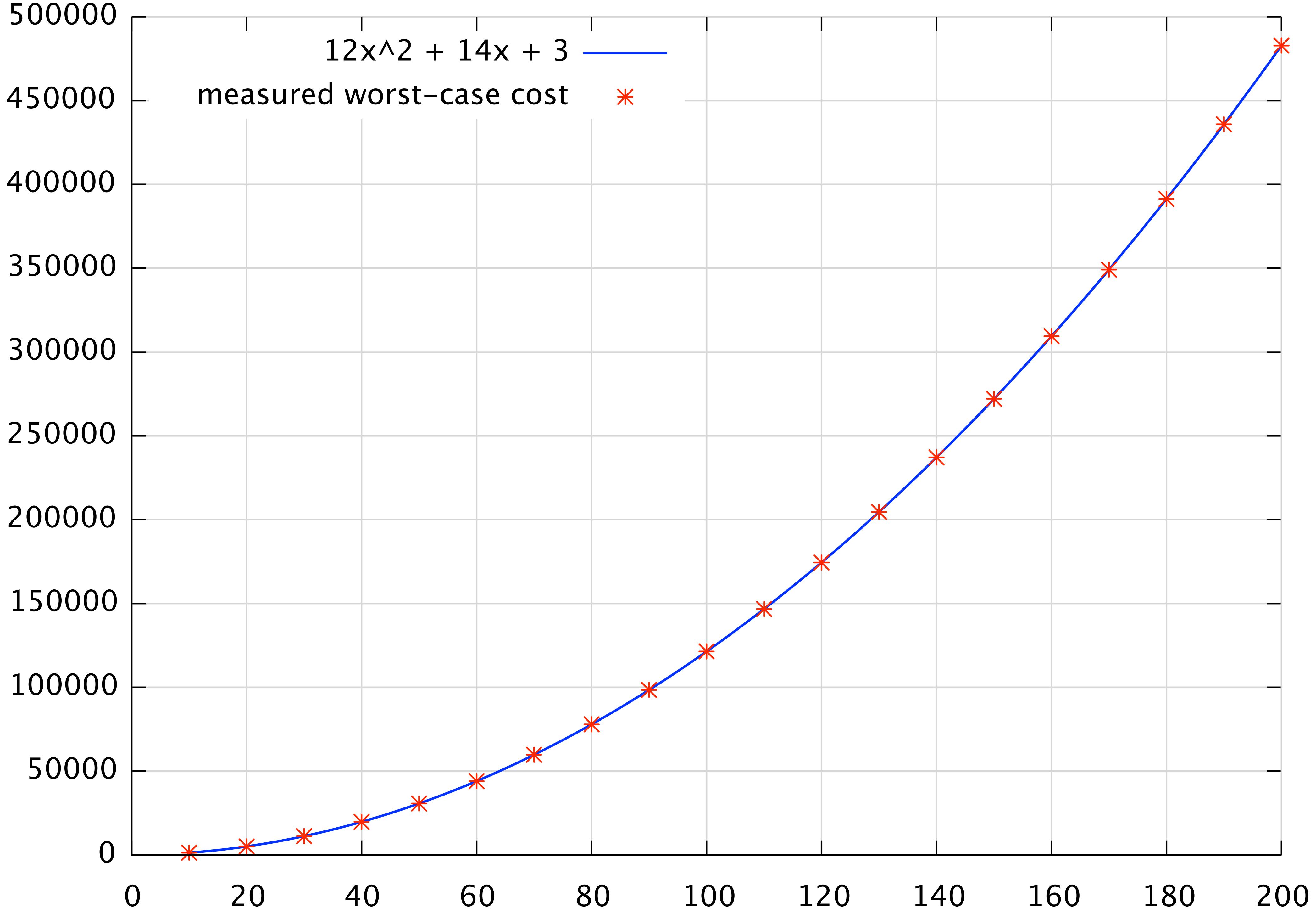
Absynth

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

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

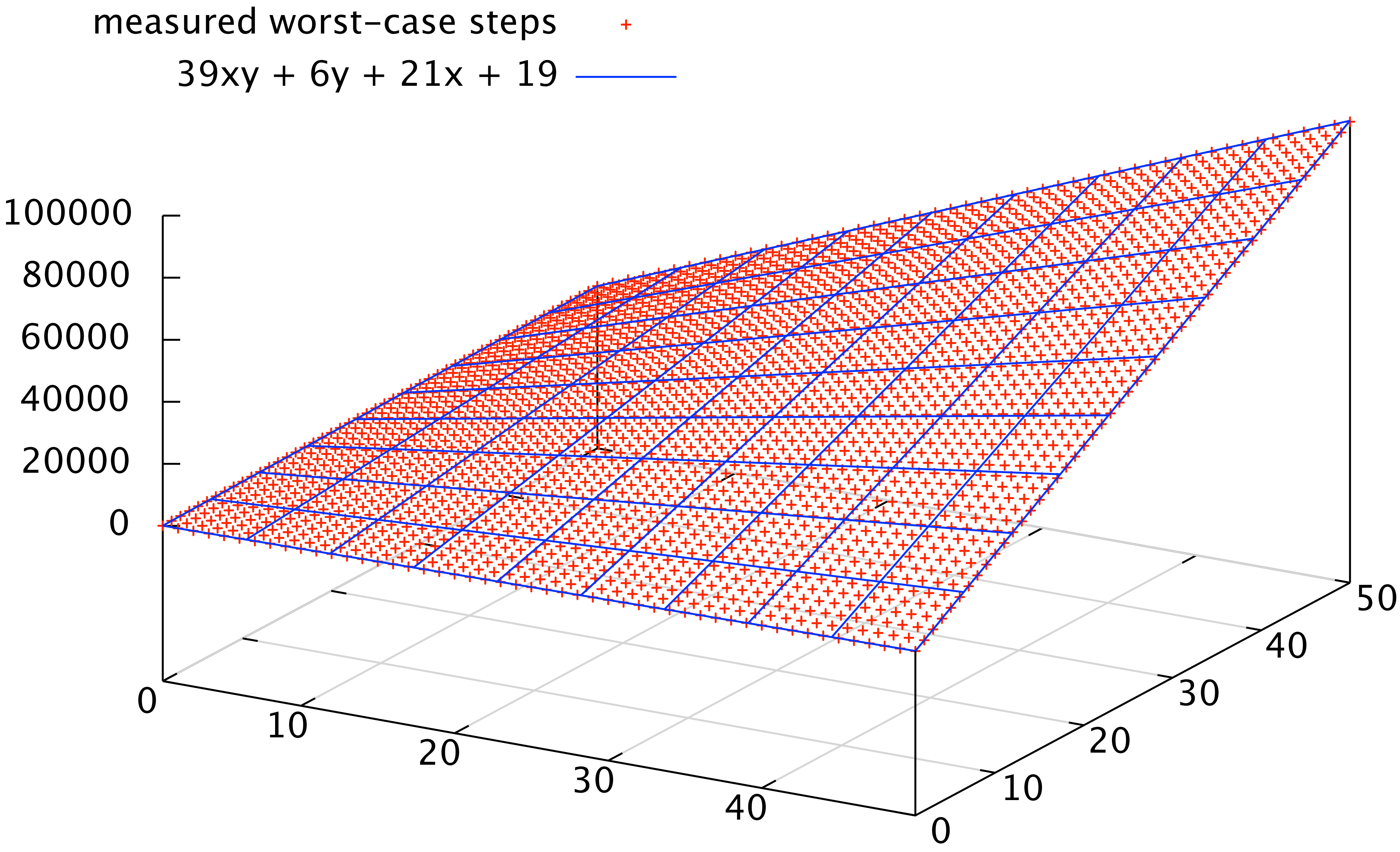
Micro Benchmarks

Evaluation-Step Bounds



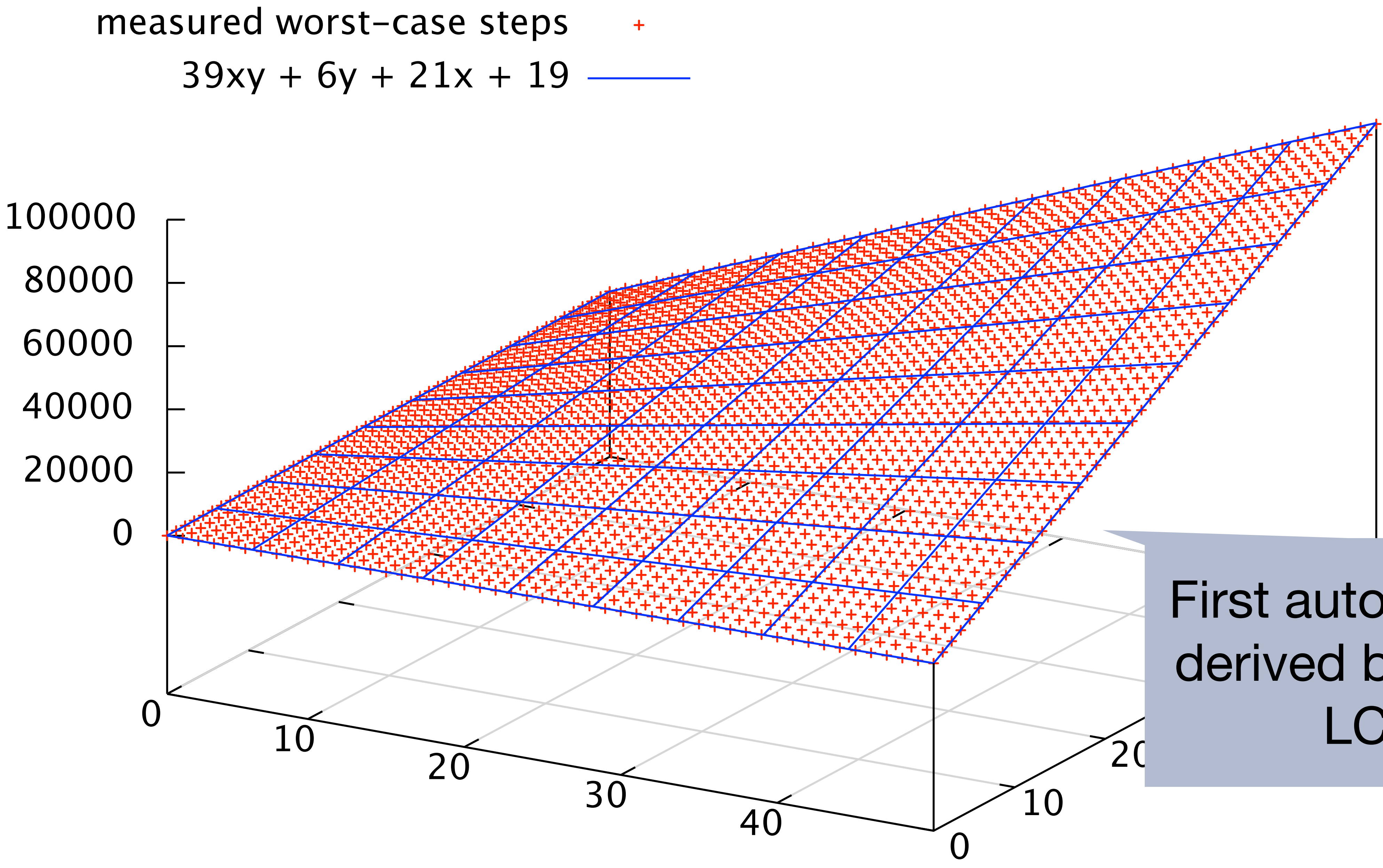
Quick Sort for Integers

Evaluation-step bound vs.
measured behavior



Longest Common Subsequence

Evaluation-step bound vs. measured behavior



First automatically derived bound for LCS.

Longest Common Subsequence

Evaluation-step bound vs. measured behavior

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:

$$\begin{aligned}\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\ \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}\end{aligned}$$

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

External choice

Example type:

$$\begin{aligned} \mathsf{queue}_{\mathbf{A}} = \&\{&\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}}, \\ &\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \\ &\qquad\quad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\} \end{aligned}$$

Internal choice

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

Receive msg of type A

Example type:

$$\begin{aligned} \mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\ & \\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\ & \\ \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\} & \end{aligned}$$

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:

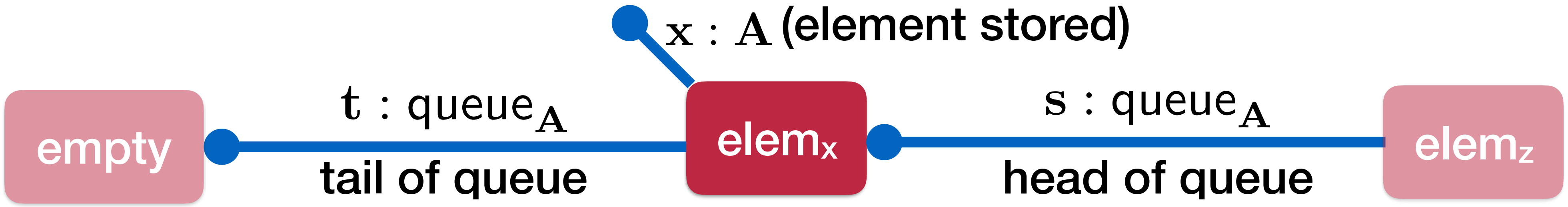
$$\begin{aligned}\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\ \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}\end{aligned}$$

Receive msg of type A

Send msg of type A

Type soundness (progress and preservation) implies deadlock freedom

Example: Queue



$\mathbf{x} : \mathbf{A}$ (element stored)

empty

$\mathbf{t}$: $\text{queue}_{\mathbf{A}}$

tail of queue

elem$_x$

$\mathbf{s}$: $\text{queue}_{\mathbf{A}}$

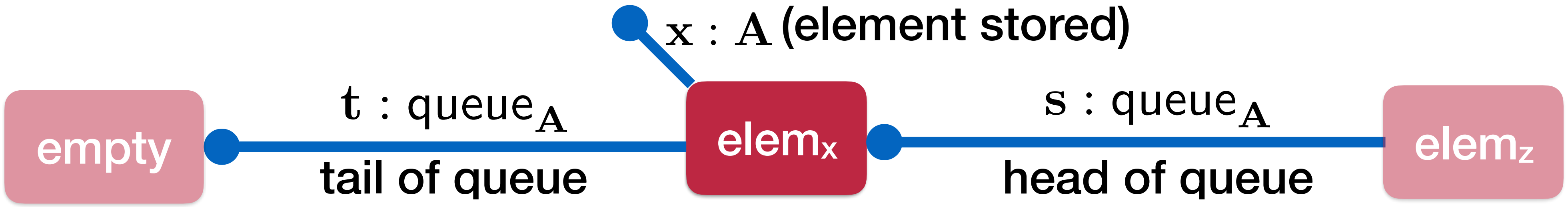
head of queue

elem$_z$

```
$(x : A)\ (t : \text{queue}_A) \vdash \mathit{elem} :: (s : \text{queue}_A)$
  $s \leftarrow \mathit{elem} \leftarrow x\ t =$
      case $s$ (ins $\Rightarrow$ $y \leftarrow$ recv $s$ ;
                     $t$.ins ;
                     send $t\ y$ ;
                     $s \leftarrow \mathit{elem} \leftarrow x\ t$
               | del $\Rightarrow$ $s$.some ;
                     send $s\ x$ ;
                     $s \leftarrow t$)
```

$$\begin{aligned} \text{queue}_{\mathbf{A}} = \&\{&\text{ins} : \mathbf{A} \multimap \text{queue}_{\mathbf{A}}, \\ &\text{del} : \oplus\{\text{none} : \mathbf{1}, \\ &\qquad \text{some} : \mathbf{A} \otimes \text{queue}_{\mathbf{A}}\}\} \end{aligned}$$

Example: Queue

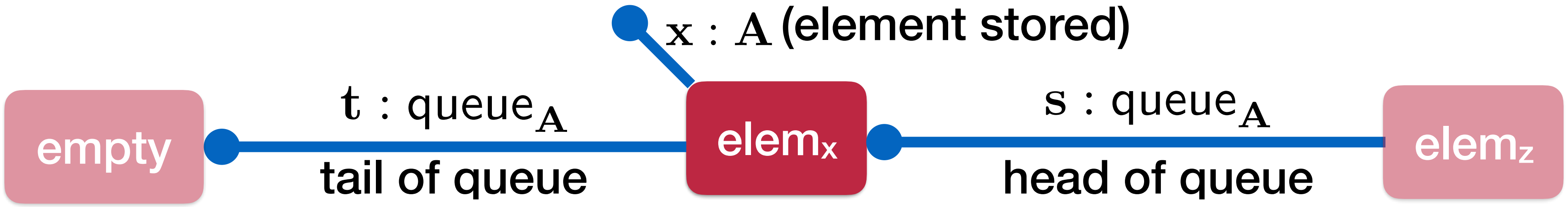


$$\text{queue}_\mathbf{A} = \&\{\text{ins} : \mathbf{A} \multimap \text{queue}_\mathbf{A}, \\ \text{del} : \oplus\{\text{none} : \mathbf{1}, \\ \text{some} : \mathbf{A} \otimes \text{queue}_\mathbf{A}\}\}$$

```
$(x : A)\ (t : \mathsf{queue}_A) \vdash elem :: (s : \mathsf{queue}_A)$
  $s \leftarrow elem \leftarrow x\ t =$
      case $s$ (ins $\Rightarrow y \leftarrow$ recv $s$ ;
                   $t$.ins ;
                   send $t\ y$ ;
                   $s \leftarrow elem \leftarrow x\ t$
              | del $\Rightarrow s$.some ;
                   send $s\ x$ ;
                   $s \leftarrow t$)
```

recv ‘ins’ and y

Example: Queue



$\mathbf{x} : \mathbf{A}$ **(element stored)**

empty

$\mathbf{t} : \mathsf{queue}_{\mathbf{A}}$

tail of queue

elem$_x$

$\mathbf{s} : \mathsf{queue}_{\mathbf{A}}$

head of queue

elem$_z$

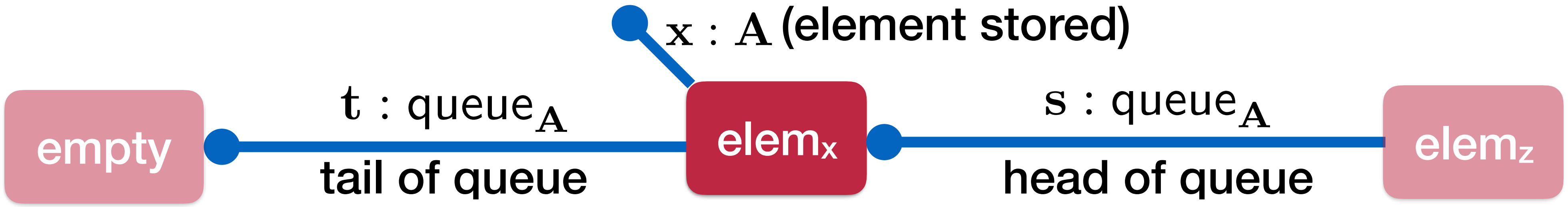
$$
\begin{aligned}
\mathsf{queue}_{\mathbf{A}} = \&\{&\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\
&\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\
&\qquad\qquad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}
\end{aligned}
$$

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

recv ‘ins’ and y

send ‘ins’ and y

Example: Queue



$$\begin{aligned}\text{queue}_{\mathbf{A}} = \&\{&\text{ins} : \mathbf{A} \multimap \text{queue}_{\mathbf{A}},\\ &\text{del} : \oplus\{\text{none} : \mathbf{1},\\ &\qquad\quad \text{some} : \mathbf{A} \otimes \text{queue}_{\mathbf{A}}\}\}\end{aligned}$$

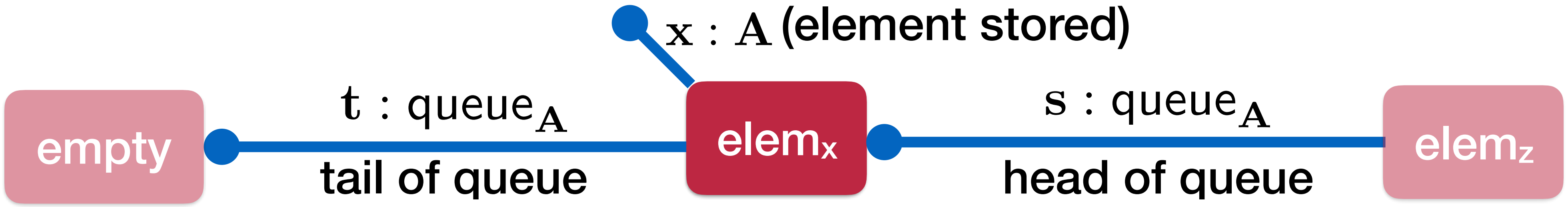
```
$(x : A)\ (t : \text{queue}_A) \vdash elem :: (s : \text{queue}_A)$
  $s \leftarrow elem \leftarrow x\ t =$
      case $s$ (ins $\Rightarrow$ $y \leftarrow$ recv $s$ ;
                    $t$.ins ;
                    send $t\ y$ ;
                    $s \leftarrow elem \leftarrow x\ t$
        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

recv ‘ins’ and y

send ‘ins’ and y

recurse

Example: Queue



$$\begin{aligned}\mathsf{queue}_A = \&\{&\mathsf{ins} : A \multimap \mathsf{queue}_A,\\ &\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\ &\qquad\mathsf{some} : A \otimes \mathsf{queue}_A\}\}\end{aligned}$$

```
$(x : A)\ (t : \mathsf{queue}_A) \vdash elem :: (s : \mathsf{queue}_A)$
  $s \leftarrow elem \leftarrow x\ t =$
      case $s$ (ins $\Rightarrow y \leftarrow$ recv $s$ ;
                    $t$.ins ;
                    send $t\ y$ ;
                    $s \leftarrow elem \leftarrow x\ t$
              | del $\Rightarrow s$.some ;
                    send $s\ x$ ;
                    $s \leftarrow t$)
```

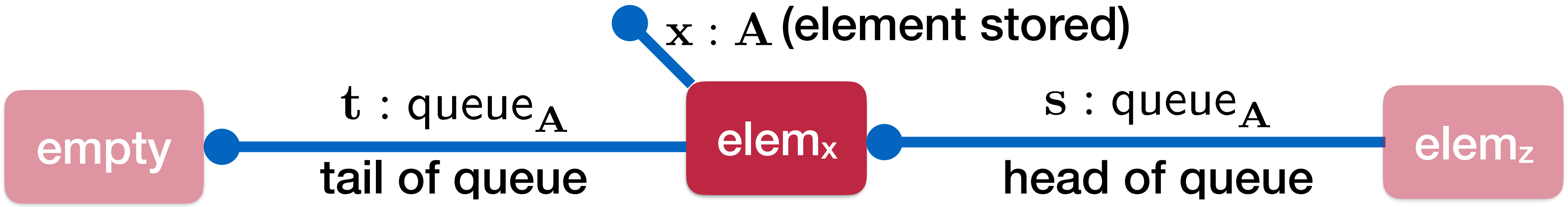
recv ‘ins’ and y

send ‘ins’ and y

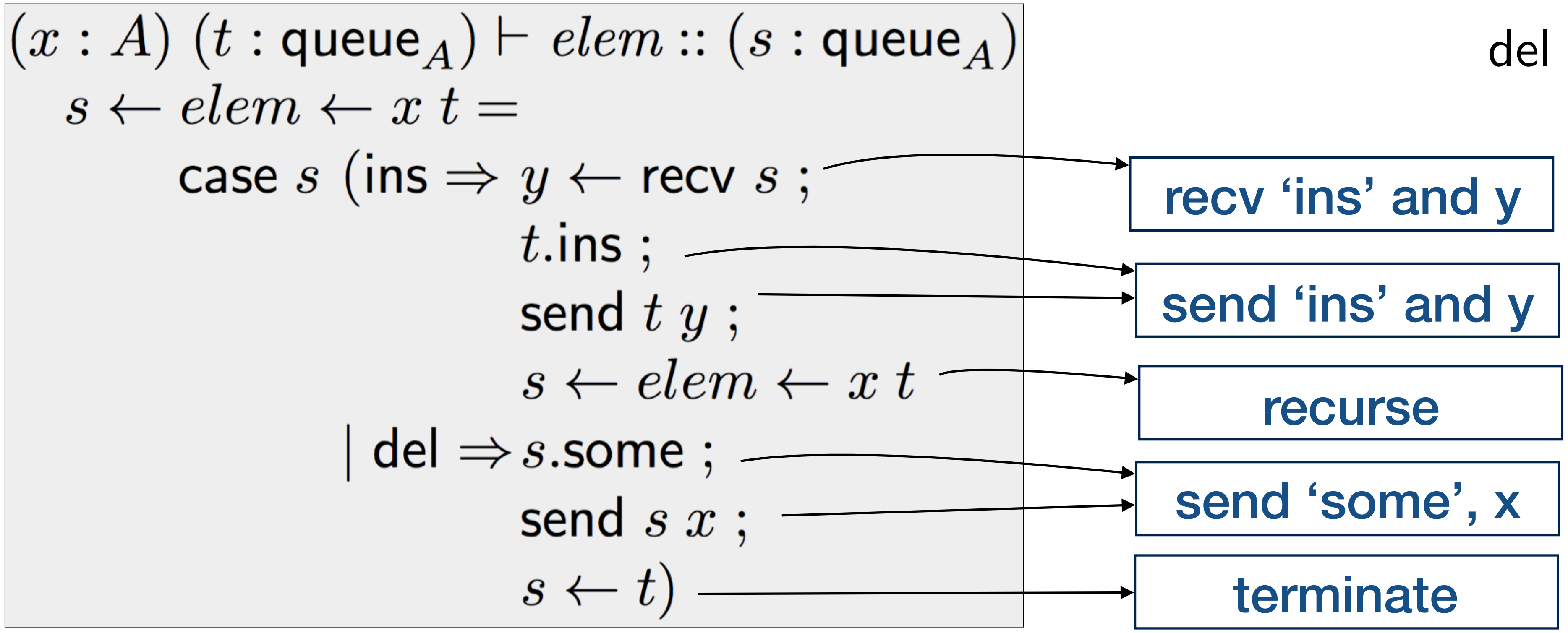
recurse

send ‘some’, x

Example: Queue



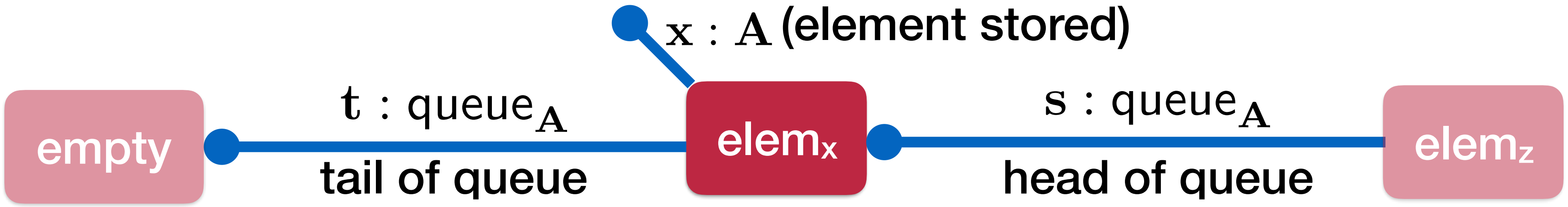
$$\mathsf{queue}_{\mathbf{A}} = \&\{\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\ \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}$$



```
$(x : A)\ (t : \mathsf{queue}_A) \vdash elem :: (s : \mathsf{queue}_A)$
  $s \leftarrow elem \leftarrow x\ t =$
      case $s$ (ins $\Rightarrow y \leftarrow$ recv $s$ ;
                    $t$.ins ;
                    send $t\ y$ ;
                    $s \leftarrow elem \leftarrow x\ t$
               | del $\Rightarrow s$.some ;
                    send $s\ x$ ;
                    $s \leftarrow t$)
```

Example: Queue

Type checking in linear time!



empty

$\mathbf{t} : \mathsf{queue}_{\mathbf{A}}$

tail of queue

elem$_x$

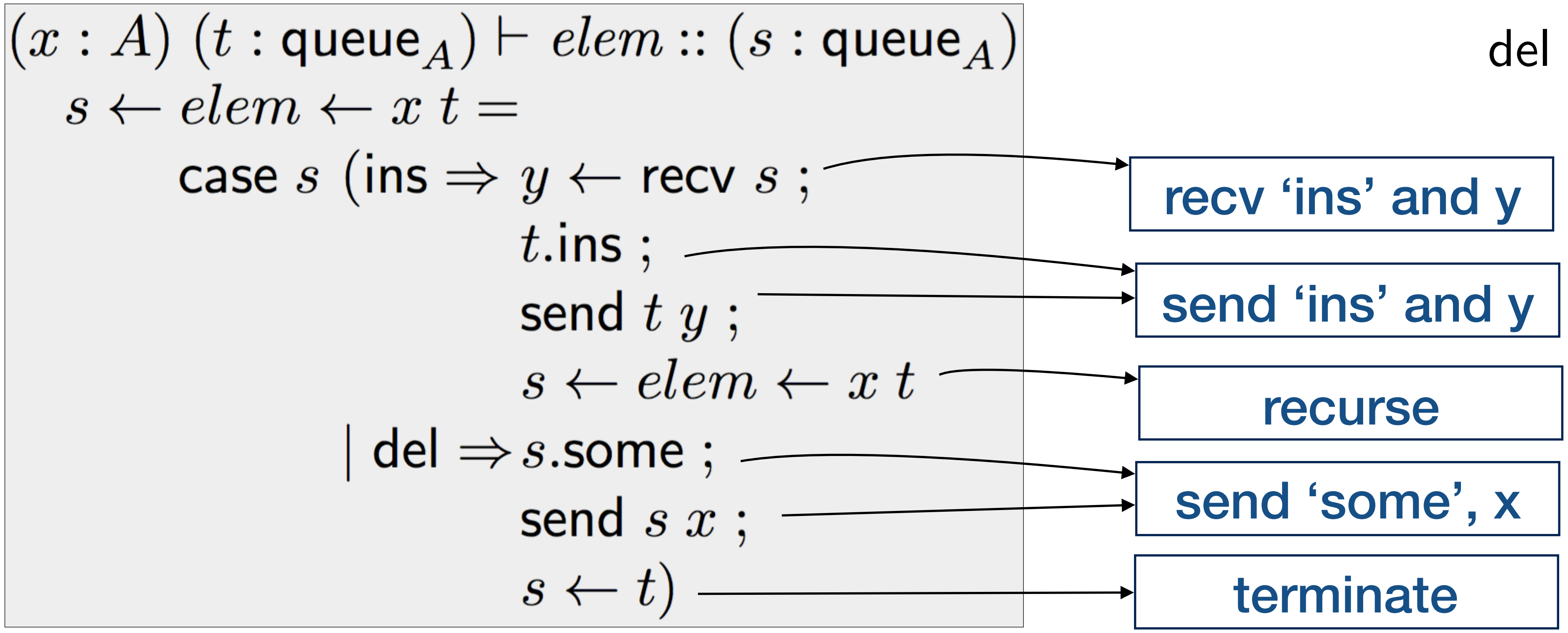
$\mathbf{x} : \mathbf{A}$ (element stored)

$\mathbf{s} : \mathsf{queue}_{\mathbf{A}}$

head of queue

elem$_z$

$$\begin{aligned}\mathsf{queue}_{\mathbf{A}} = \&\{&\mathsf{ins} : \mathbf{A} \multimap \mathsf{queue}_{\mathbf{A}},\\ &\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1},\\ &\qquad \mathsf{some} : \mathbf{A} \otimes \mathsf{queue}_{\mathbf{A}}\}\}\end{aligned}$$



```
$(x : A)\ (t : \mathsf{queue}_A) \vdash elem :: (s : \mathsf{queue}_A)$
  $s \leftarrow elem \leftarrow x\ t =$
      case $s$ (ins $\Rightarrow$ $y \leftarrow$ recv $s$ ;
                $t$.ins ;
                send $t\ y$ ;
                $s \leftarrow elem \leftarrow x\ t$
            | del $\Rightarrow$ $s$.some ;
                send $s\ x$ ;
                $s \leftarrow t$)
```

recv ‘ins’ and y

send ‘ins’ and y

recurse

send ‘some’, x

terminate

Example: Auction

$$
\begin{aligned}
\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}
\end{aligned}
$$

Example: Auction

sends status of auction

$$\begin{aligned}\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\ &\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}\end{aligned}$$

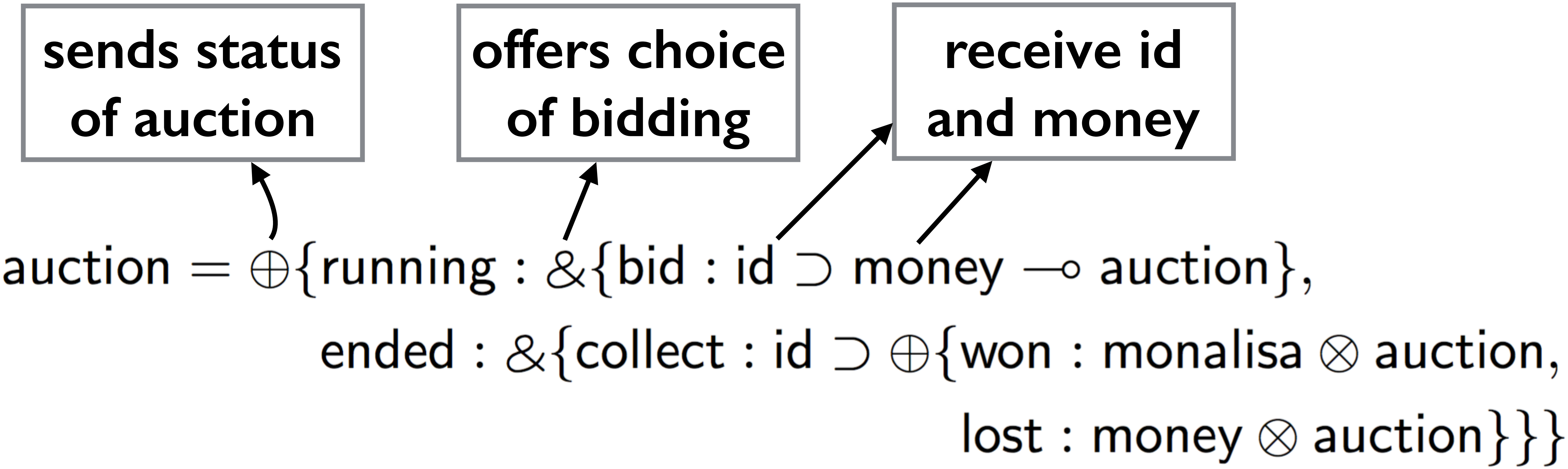
Example: Auction

sends status of auction

offers choice of bidding

$$
\begin{aligned}
\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}
\end{aligned}
$$

Example: Auction



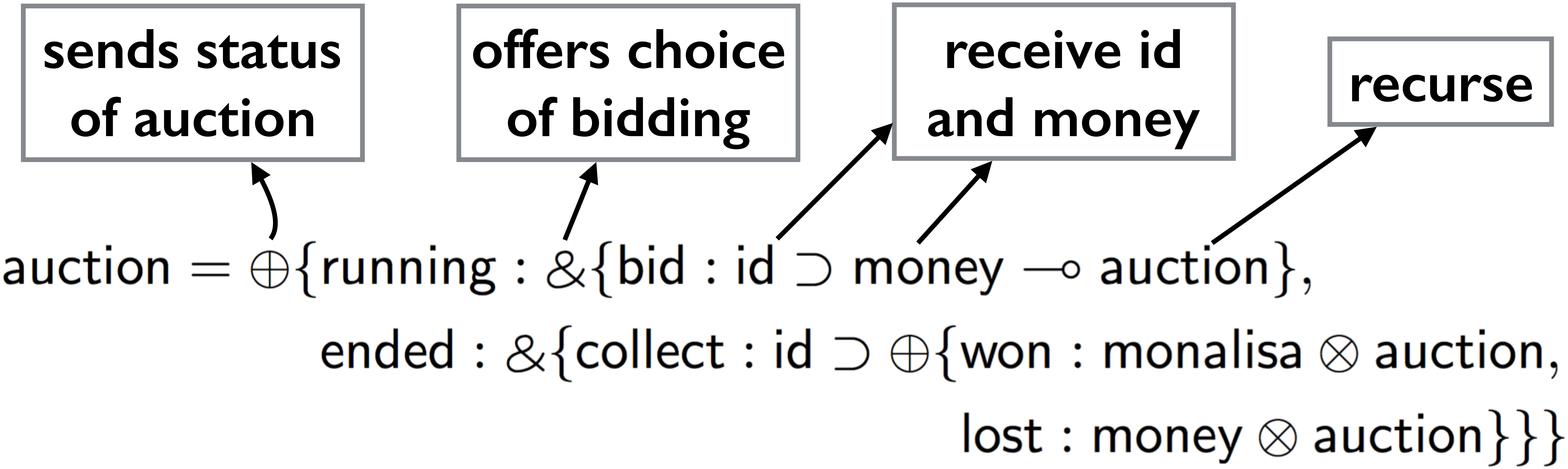
sends status of auction

offers choice of bidding

receive id and money

$$\begin{aligned}\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\ &\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}\end{aligned}$$

Example: Auction



sends status of auction

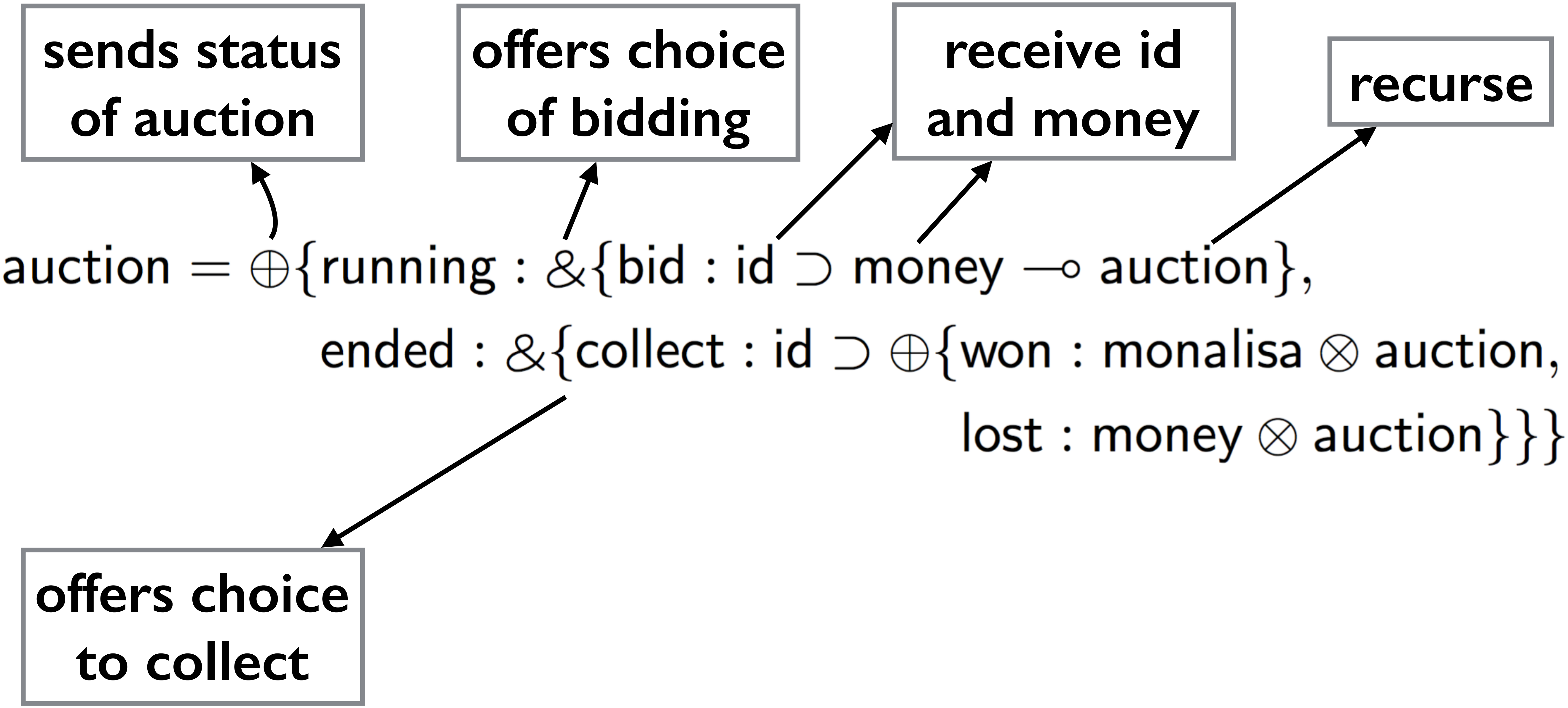
offers choice of bidding

receive id and money

recurse

$$\begin{aligned}
\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}
\end{aligned}$$

Example: Auction



sends status of auction

offers choice of bidding

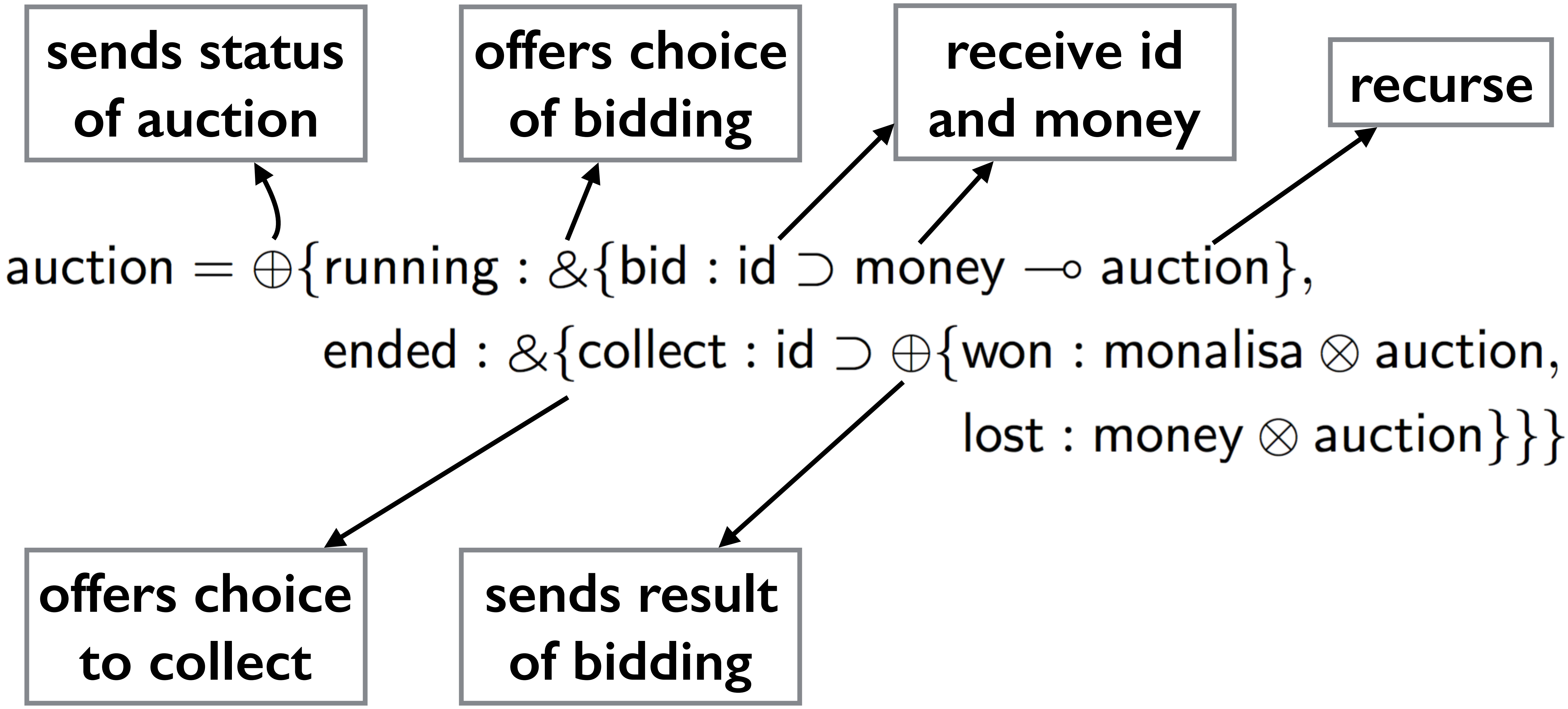
receive id and money

recurse

$$
\begin{aligned}
\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}
\end{aligned}
$$

offers choice to collect

Example: Auction



sends status of auction

offers choice of bidding

receive id and money

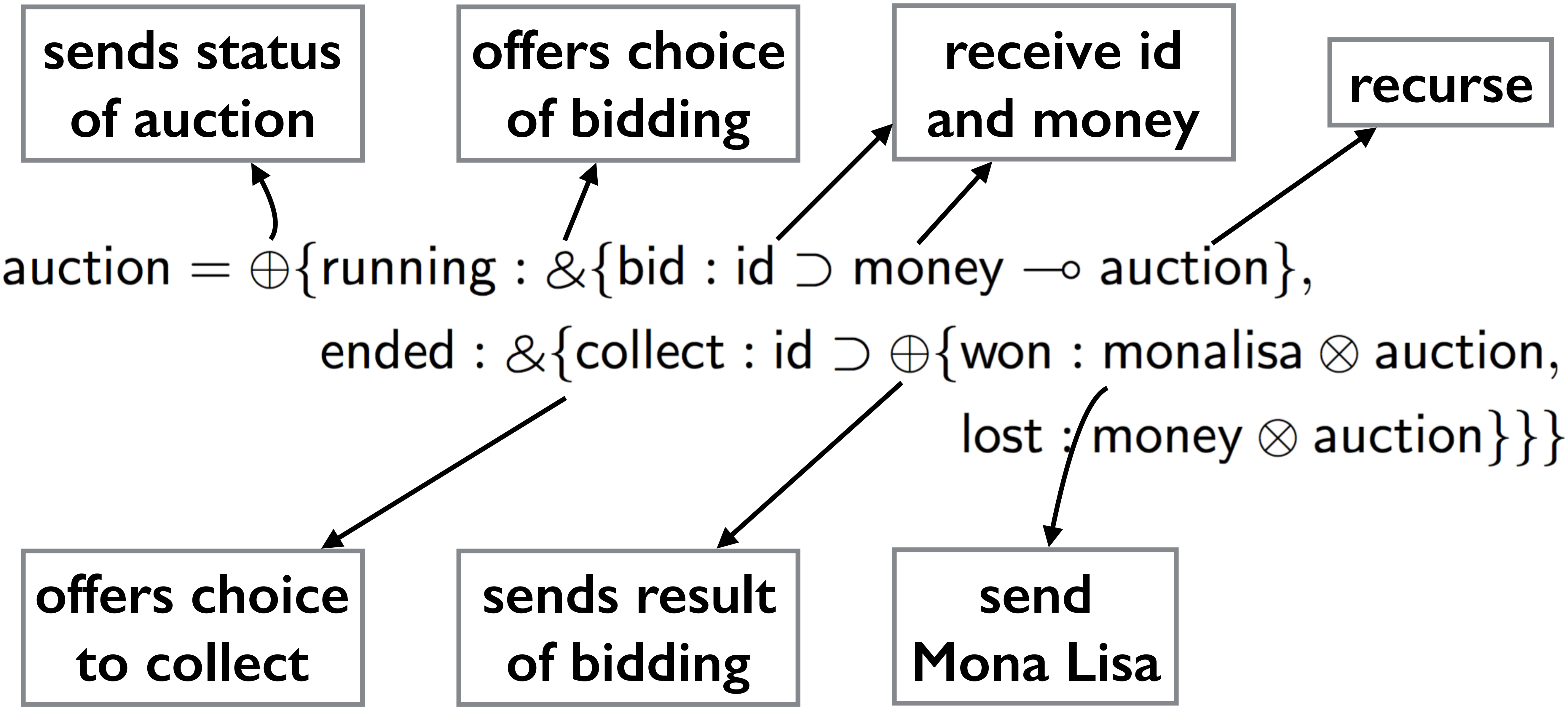
recurse

$$\begin{aligned}\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\ &\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}\end{aligned}$$

offers choice to collect

sends result of bidding

Example: Auction



sends status of auction

offers choice of bidding

receive id and money

recurse

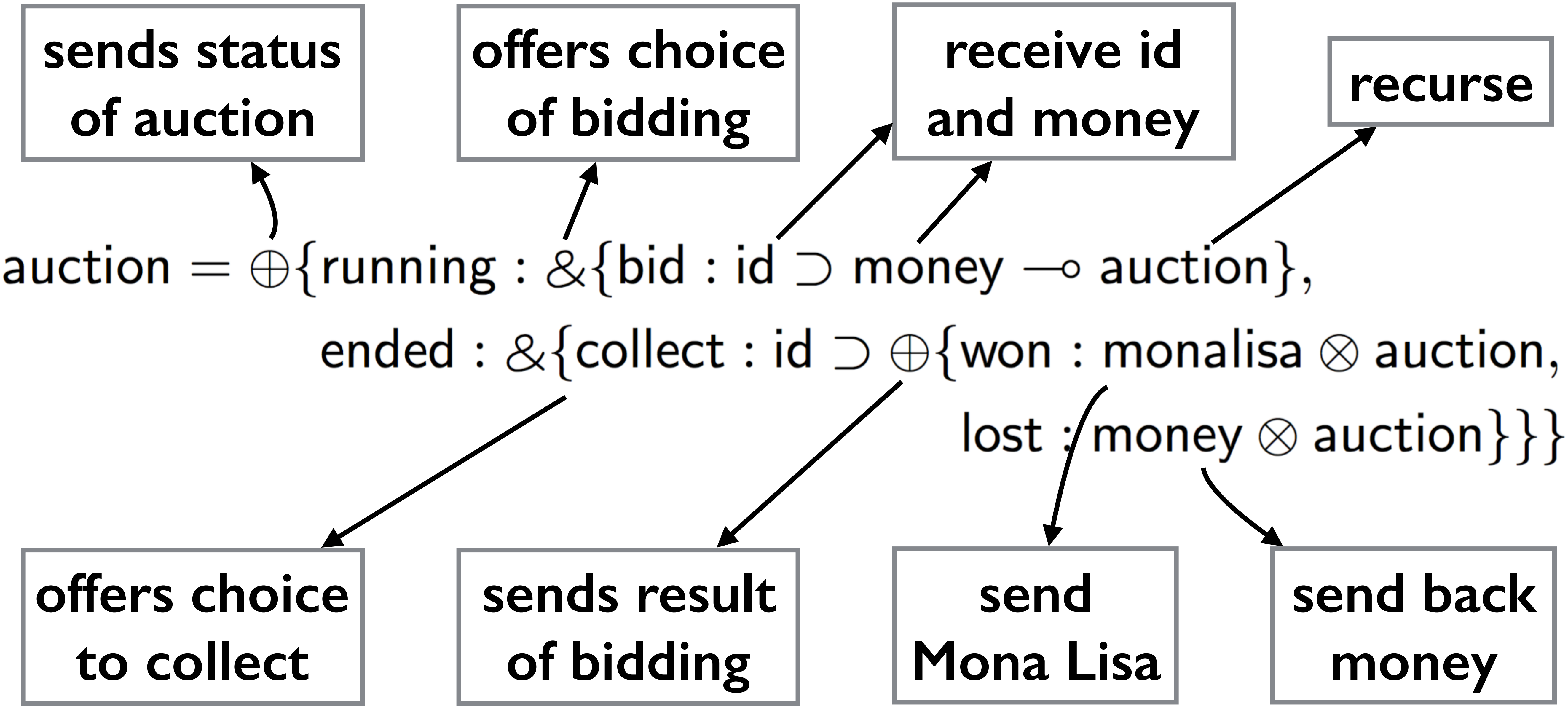
$$
\begin{aligned}
\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}
\end{aligned}
$$

offers choice to collect

sends result of bidding

send Mona Lisa

Example: Auction



sends status of auction

offers choice of bidding

receive id and money

recurse

$$\begin{aligned}\mathsf{auction} = \oplus\{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \mathsf{auction}\},\\ &\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \oplus\{\mathsf{won} : \mathsf{monalisa} \otimes \mathsf{auction},\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes \mathsf{auction}\}\}\}\end{aligned}$$

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***

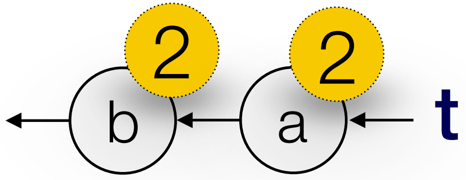
Potential transfer only at the type level, not at runtime.

User-defined cost metric.

- Message potential is a function of (functional) payload

$$\begin{array}{rll} A, B, C ::= & \tau \supset A & \text{input value of type } \tau \text{ and continue as } A \\ \mid & \tau \wedge A & \text{output value of type } \tau \text{ and continue as } A \end{array}$$

$\texttt{L}^{2}\texttt{(int)}$



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***

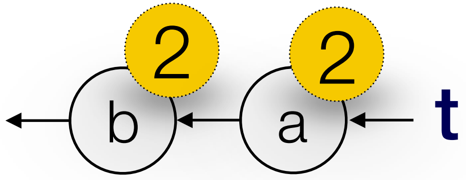
Potential transfer only at the type level, not at runtime.

User-defined cost metric.

- Message potential is a function of (functional) payload

$$\begin{array}{llll} A, B, C & ::= & \tau \supset A & \text{input value of type } \tau \text{ and continue as } A \\ & \mid & \tau \wedge A & \text{output value of type } \tau \text{ and continue as } A \end{array}$$

$L^2(\texttt{int})$



- Syntactic sugar (no payload)

$$A ::= \ldots \mid \triangleright^r A \mid \triangleleft^r A$$

- Only in intermediate language:

$$\text{get } x_m \; \{r\} \; ; \; P$$

$$\text{pay } x_m \; \{r\} \; ; \; P$$

Resource-Aware Session Types

Efficient type inference via LP solving

- Each ***process stores potential*** in functional data
- Potential can be ***transferred via messages***

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

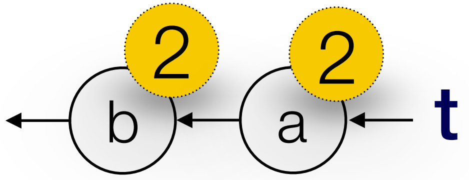
- Potential is used to ***pay for performed work***

User-defined cost metric.

- Message potential is a function of (functional) payload

$$\begin{array}{rcll} A, B, C & ::= & \tau \supset A & \text{input value of type } \tau \text{ and continue as } A \\ & \mid & \tau \wedge A & \text{output value of type } \tau \text{ and continue as } A \end{array}$$

$L^{2}(\mathtt{int})$



- Syntactic sugar (no payload)

$$A ::= \ldots \mid \triangleright^{r} A \mid \triangleleft^{r} A$$

- Only in intermediate language:

$$\text{get } x_m \; \{r\} \; ; \; P$$

$$\text{pay } x_m \; \{r\} \; ; \; P$$

$$
\begin{aligned}
\mathsf{auction} = {\uparrow}_{\mathsf{L}}^{\mathsf{S}}\ \triangleleft^{11} \oplus \{ \mathsf{running} : \&\{ & \mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1}\ {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\ \mathsf{auction}, \\
& \mathsf{cancel} : \triangleright^{8}\ {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\ \mathsf{auction}\}, \\
\mathsf{ended} : \&\{ & \mathsf{collect} : \mathsf{id} \supset \\
& \quad \oplus\{ \mathsf{won} : \mathsf{lot} \otimes \triangleright^{3}\ {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\ \mathsf{auction}, \\
& \quad\quad \mathsf{lost} : \mathsf{money} \otimes {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\ \mathsf{auction}\}, \\
& \mathsf{cancel} : \triangleright^{8}\ {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\ \mathsf{auction}\}\}
\end{aligned}
$$

Example: Type of an Auction Contract

Example: Type of an Auction Contract

Sharing: Need to acquire contract before use.

$$
\begin{aligned}
\mathsf{auction} = {\uparrow}_{L}^{S} \triangleleft^{11} \oplus \{ & \mathsf{running} : \& \{ \mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} {\downarrow}_{L}^{S} \mathsf{auction}, \\
& \qquad \mathsf{cancel} : \triangleright^{8} {\downarrow}_{L}^{S} \mathsf{auction} \}, \\
& \mathsf{ended} : \& \{ \mathsf{collect} : \mathsf{id} \supset \\
& \qquad \oplus \{ \mathsf{won} : \mathsf{lot} \otimes \triangleright^{3} {\downarrow}_{L}^{S} \mathsf{auction}, \\
& \qquad \quad \mathsf{lost} : \mathsf{money} \otimes {\downarrow}_{L}^{S} \mathsf{auction} \}, \\
& \qquad \mathsf{cancel} : \triangleright^{8} {\downarrow}_{L}^{S} \mathsf{auction} \} \}
\end{aligned}
$$

Sharing: Need to acquire contract before use.

Equi-synchronizing: Release contract at the same type.

$$\begin{aligned}
\mathrm{auction} = {\uparrow}_{\mathsf{L}}^{\mathsf{S}} \triangleleft^{11} \oplus \{&\mathrm{running} : \&\{\mathrm{bid} : \mathrm{id} \supset \mathrm{money} \multimap \triangleright^{1} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction}, \\
&\qquad\qquad \mathrm{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction}\}, \\
&\mathrm{ended} : \&\{\mathrm{collect} : \mathrm{id} \supset \\
&\qquad\qquad \oplus\{\mathrm{won} : \mathrm{lot} \otimes \triangleright^{3} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction}, \\
&\qquad\qquad\quad \mathrm{lost} : \mathrm{money} \otimes {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction}\}, \\
&\qquad\quad \mathrm{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction}\}\}
\end{aligned}$$

Example: Type of an Auction Contract

Example: Type of an Auction Contract

$$
\begin{aligned}
\mathsf{auction} = {\uparrow}_{\mathsf{L}}^{\mathsf{S}}\, \triangleleft^{11} \oplus \{ \mathsf{running} : \&\{ & \mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}, \\
& \mathsf{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}, \\
\mathsf{ended} : \&\{ & \mathsf{collect} : \mathsf{id} \supset \\
& \quad \oplus\{ \mathsf{won} : \mathsf{lot} \otimes \triangleright^{3} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}, \\
& \quad \phantom{\oplus\{} \mathsf{lost} : \mathsf{money} \otimes {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}, \\
& \mathsf{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}\}
\end{aligned}
$$

Action can be open (running) or closed (ended).

Example: Type of an Auction Contract

Sending a functional value.

$$\begin{aligned}
\mathsf{auction} = {\uparrow}_{\mathsf{L}}^{\mathsf{S}}\, {\triangleleft}^{11} \oplus \{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap {\triangleright}^{1}\, {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction},\\
&\qquad\qquad\qquad \mathsf{cancel} : {\triangleright}^{8}\, {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\},\\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset\\
&\qquad\qquad\qquad \oplus\{\mathsf{won} : \mathsf{lot} \otimes {\triangleright}^{3}\, {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction},\\
&\qquad\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\},\\
&\qquad\qquad \mathsf{cancel} : {\triangleright}^{8}\, {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}\}
\end{aligned}$$

Action can be open (running) or closed (ended).

Example: Type of an Auction Contract

Sending a functional value.

Sending a linear value.

$$
\begin{aligned}
\mathsf{auction} = {\uparrow}_{\mathsf{L}}^{\mathsf{S}} \triangleleft^{11} \oplus \{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}, \\
&\qquad\qquad \mathsf{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}, \\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \\
&\qquad\qquad \oplus\{\mathsf{won} : \mathsf{lot} \otimes \triangleright^{3} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}, \\
&\qquad\qquad \quad \mathsf{lost} : \mathsf{money} \otimes {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}, \\
&\qquad\quad \mathsf{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}\}
\end{aligned}
$$

Action can be open (running) or closed (ended).

Example: Type of an Auction Contract

Sending a functional value.

Sending a linear value.

$$\begin{aligned}
\mathsf{auction} = {\uparrow}_{\mathsf{L}}^{\mathsf{S}} \triangleleft^{11} \oplus \{ &\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}, \\
&\qquad\qquad \mathsf{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}, \\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \\
&\qquad\qquad \oplus\{\mathsf{won} : \mathsf{lot} \otimes \triangleright^{3} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}, \\
&\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}, \\
&\qquad \mathsf{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}\}
\end{aligned}$$

Action can be open (running) or closed (ended).

Can collect lot or reclaim your bid.

$$
\begin{aligned}
\mathsf{auction} = \uparrow_{\mathsf{L}}^{\mathsf{S}} \lhd^{11} \oplus \{ &\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \rhd^{1} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}, \\
&\qquad\qquad\qquad\quad \mathsf{cancel} : \rhd^{8} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}, \\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \\
&\qquad\qquad\qquad \oplus\{\mathsf{won} : \mathsf{lot} \otimes \rhd^{3} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}, \\
&\qquad\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}, \\
&\qquad\qquad \mathsf{cancel} : \rhd^{8} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\}\}
\end{aligned}
$$

Example: Type of an Auction Contract

Example: Type of an Auction Contract

At the beginning, you have to pay 11 units to cover the worst-case gas cost.

$$
\begin{aligned}
\text{auction} = \uparrow_{L}^{S} \triangleleft^{11} \oplus \{ & \text{running} : \&\{\text{bid} : \text{id} \supset \text{money} \multimap \triangleright^{1} \downarrow_{L}^{S} \text{auction}, \\
& \qquad \text{cancel} : \triangleright^{8} \downarrow_{L}^{S} \text{auction}\}, \\
& \text{ended} : \&\{\text{collect} : \text{id} \supset \\
& \qquad \oplus\{\text{won} : \text{lot} \otimes \triangleright^{3} \downarrow_{L}^{S} \text{auction}, \\
& \qquad \quad \text{lost} : \text{money} \otimes \downarrow_{L}^{S} \text{auction}\}, \\
& \quad \text{cancel} : \triangleright^{8} \downarrow_{L}^{S} \text{auction}\}\}
\end{aligned}
$$

Example: Type of an Auction Contract

At the beginning, you have to pay 11 units to cover the worst-case gas cost.

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

$$\begin{aligned}
\mathsf{auction} = {\uparrow}_{\mathsf{L}}^{\mathsf{S}}\, \triangleleft^{11} \oplus \{&\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction},\\
&\qquad\qquad \mathsf{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\},\\
&\mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset\\
&\qquad\qquad \oplus\{\mathsf{won} : \mathsf{lot} \otimes \triangleright^{3} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction},\\
&\qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\},\\
&\qquad\quad \mathsf{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathsf{auction}\}\}
\end{aligned}$$

Example: Type of an Auction Contract

At the beginning, you have to pay 11 units to cover the worst-case gas cost.

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

$$
\begin{aligned}
\mathsf{auction} = {\uparrow}^{\mathsf{S}}_{\mathsf{L}}\, {\color{red}\triangleleft^{11}} \oplus \{ & \mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap {\color{red}\triangleright^{1}}\, {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\, \mathsf{auction}, \\
& \qquad\qquad\qquad \mathsf{cancel} : {\color{red}\triangleright^{8}}\, {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\, \mathsf{auction}\}, \\
& \mathsf{ended} : \&\{\mathsf{collect} : \mathsf{id} \supset \\
& \qquad\qquad \oplus\{\mathsf{won} : \mathsf{lot} \otimes {\color{red}\triangleright^{3}}\, {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\, \mathsf{auction}, \\
& \qquad\qquad\quad \mathsf{lost} : \mathsf{money} \otimes {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\, \mathsf{auction}\}, \\
& \qquad\quad \mathsf{cancel} : {\color{red}\triangleright^{8}}\, {\downarrow}^{\mathsf{S}}_{\mathsf{L}}\, \mathsf{auction}\}\}
\end{aligned}
$$

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

At the beginning, you have to pay 11 units to cover the worst-case gas cost.

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

$$
\begin{aligned}
\mathrm{auction} = {\uparrow}_{\mathsf{L}}^{\mathsf{S}} \triangleleft^{11} \oplus \{ & \mathrm{running} : \& \{ \mathrm{bid} : \mathrm{id} \supset \mathrm{money} \multimap \triangleright^{1} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction}, \\
& \qquad \mathrm{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction} \}, \\
& \mathrm{ended} : \& \{ \mathrm{collect} : \mathrm{id} \supset \\
& \qquad \oplus \{ \mathrm{won} : \mathrm{lot} \otimes \triangleright^{3} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction}, \\
& \qquad\quad \mathrm{lost} : \mathrm{money} \otimes {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction} \}, \\
& \qquad \mathrm{cancel} : \triangleright^{8} {\downarrow}_{\mathsf{L}}^{\mathsf{S}}\, \mathrm{auction} \} \}
\end{aligned}
$$

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

This is the worst-case path

Example: Type of an Auction Contract

Implementation of a Running Auction

$$\mathsf{auction} = \uparrow_{\mathsf{L}}^{\mathsf{S}} \triangleleft^{11} \oplus \{\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\},$$

```
$(b : \mathsf{bids})\ ;\ (M : \mathsf{money}), (ml : \mathsf{monalisa}) \vdash \textit{run} :: (sa : \mathsf{auction})$
  $sa \leftarrow \textit{run}\ b \leftarrow M\ l =$
    $la \leftarrow \mathsf{accept}\ sa\ ;$
    $la.\mathsf{running}\ ;$
    $\mathsf{case}\ la$
      $(\mathsf{bid} \Rightarrow r \leftarrow \mathsf{recv}\ la\ ;$
                $m \leftarrow \mathsf{recv}\ la\ ;$
                $sa \leftarrow \mathsf{detach}\ la\ ;$
                $m.\mathsf{value}\ ;$
                $v \leftarrow \mathsf{recv}\ m\ ;$
                $b' = \textit{addbid}\ b\ (r, v)\ ;$
                $M' \leftarrow \textit{add} \leftarrow M\ m\ ;$
                $sa \leftarrow \textit{run}\ b' \leftarrow M'\ ml)$
```

Implementation of a Running Auction

$$\mathsf{auction} = \underline{\uparrow_{\mathsf{L}}^{\mathsf{S}}} \triangleleft^{11} \oplus \{\mathsf{running} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} \downarrow_{\mathsf{L}}^{\mathsf{S}} \mathsf{auction}\},$$

```
$(b : \mathsf{bids})\ ;\ (M : \mathsf{money}), (ml : \mathsf{monalisa}) \vdash \mathit{run} :: (sa : \mathsf{auction})$
  $sa \leftarrow \mathit{run}\ b \leftarrow M\ l =$
    $la \leftarrow \mathsf{accept}\ sa\ ;$
    $la.\mathsf{running}\ ;$
    $\mathsf{case}\ la$
      $(\mathsf{bid} \Rightarrow r \leftarrow \mathsf{recv}\ la\ ;$
                $m \leftarrow \mathsf{recv}\ la\ ;$
                $sa \leftarrow \mathsf{detach}\ la\ ;$
                $m.\mathsf{value}\ ;$
                $v \leftarrow \mathsf{recv}\ m\ ;$
                $b' = \mathit{addbid}\ b\ (r, v)\ ;$
                $M' \leftarrow \mathit{add} \leftarrow M\ m\ ;$
                $sa \leftarrow \mathit{run}\ b' \leftarrow M'\ ml)$
```

accept 'acquire' ($\uparrow_{\mathsf{L}}^{\mathsf{S}}$)

Implementation of a Running Auction

$$\mathsf{auction} = \underline{\uparrow_{\mathsf{L}}^{\mathsf{S}}}\triangleleft^{11} \oplus \{\underline{\mathsf{running}} : \&\{\mathsf{bid} : \mathsf{id} \supset \mathsf{money} \multimap \triangleright^{1} \downarrow_{\mathsf{L}}^{\mathsf{S}}\mathsf{auction}\},$$

```
$(b : \mathsf{bids})\ ;\ (M : \mathsf{money}), (ml : \mathsf{monalisa}) \vdash \textit{run} :: (sa : \mathsf{auction})$
  $sa \leftarrow \textit{run}\ b \leftarrow M\ l =$
    $la \leftarrow \mathsf{accept}\ sa\ ;$
    $la.\mathsf{running}\ ;$
    $\mathsf{case}\ la$
      $(\mathsf{bid} \Rightarrow r \leftarrow \mathsf{recv}\ la\ ;$
                $m \leftarrow \mathsf{recv}\ la\ ;$
                $sa \leftarrow \mathsf{detach}\ la\ ;$
                $m.\mathsf{value}\ ;$
                $v \leftarrow \mathsf{recv}\ m\ ;$
                $b' = \textit{addbid}\ b\ (r, v)\ ;$
                $M' \leftarrow \textit{add} \leftarrow M\ m\ ;$
                $sa \leftarrow \textit{run}\ b' \leftarrow M'\ ml)$
```

accept ‘acquire’ ($\uparrow_{\mathsf{L}}^{\mathsf{S}}$)

send status ‘running’

Implementation of a Running Auction

$$\mathsf{auction} = \underline{\uparrow^{\mathsf{S}}_{\mathsf{L}}} \triangleleft^{11} \oplus \{\underline{\mathsf{running}} : \&\{\mathsf{bid} : \underline{\mathsf{id}} \supset \underline{\mathsf{money}} \multimap \triangleright^{1} \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{auction}\},$$

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

```
sa ← run b ← M l =
  la ← accept sa ;
  la.running ;
  case la
    (bid ⇒ r ← recv la ;
           m ← recv la ;
           sa ← detach la ;
           m.value ;
           v ← recv m ;
           b′ = addbid b (r, v) ;
           M′ ← add ← M m ;
           sa ← run b′ ← M′ ml)
```

accept ‘acquire’ ($\uparrow^{\mathsf{S}}_{\mathsf{L}}$)

send status ‘running’

recv ‘id’ and ‘money’

Implementation of a Running Auction

$$\mathsf{auction} = \underline{\uparrow^{\mathsf{S}}_{\mathsf{L}}}\triangleleft^{11} \oplus \underline{\{\mathsf{running}} : \&\{\mathsf{bid} : \underline{\mathsf{id}} \supset \underline{\mathsf{money}} \multimap \triangleright^{1} \underline{\downarrow^{\mathsf{S}}_{\mathsf{L}}}\mathsf{auction}\},$$

```
$(b : \mathsf{bids})\ ;\ (M : \mathsf{money}), (ml : \mathsf{monalisa}) \vdash run :: (sa : \mathsf{auction})$
  $sa \leftarrow run\ b \leftarrow M\ l =$
    $la \leftarrow \mathsf{accept}\ sa$ ;
    $la.\mathsf{running}$ ;
    $\mathsf{case}\ la$
      $(\mathsf{bid} \Rightarrow r \leftarrow \mathsf{recv}\ la$ ;
                $m \leftarrow \mathsf{recv}\ la$ ;
                $sa \leftarrow \mathsf{detach}\ la$ ;
                $m.\mathsf{value}$ ;
                $v \leftarrow \mathsf{recv}\ m$ ;
                $b' = addbid\ b\ (r, v)$ ;
                $M' \leftarrow add \leftarrow M\ m$ ;
                $sa \leftarrow run\ b' \leftarrow M'\ ml)$
```

accept ‘acquire’ ($\uparrow^{\mathsf{S}}_{\mathsf{L}}$)

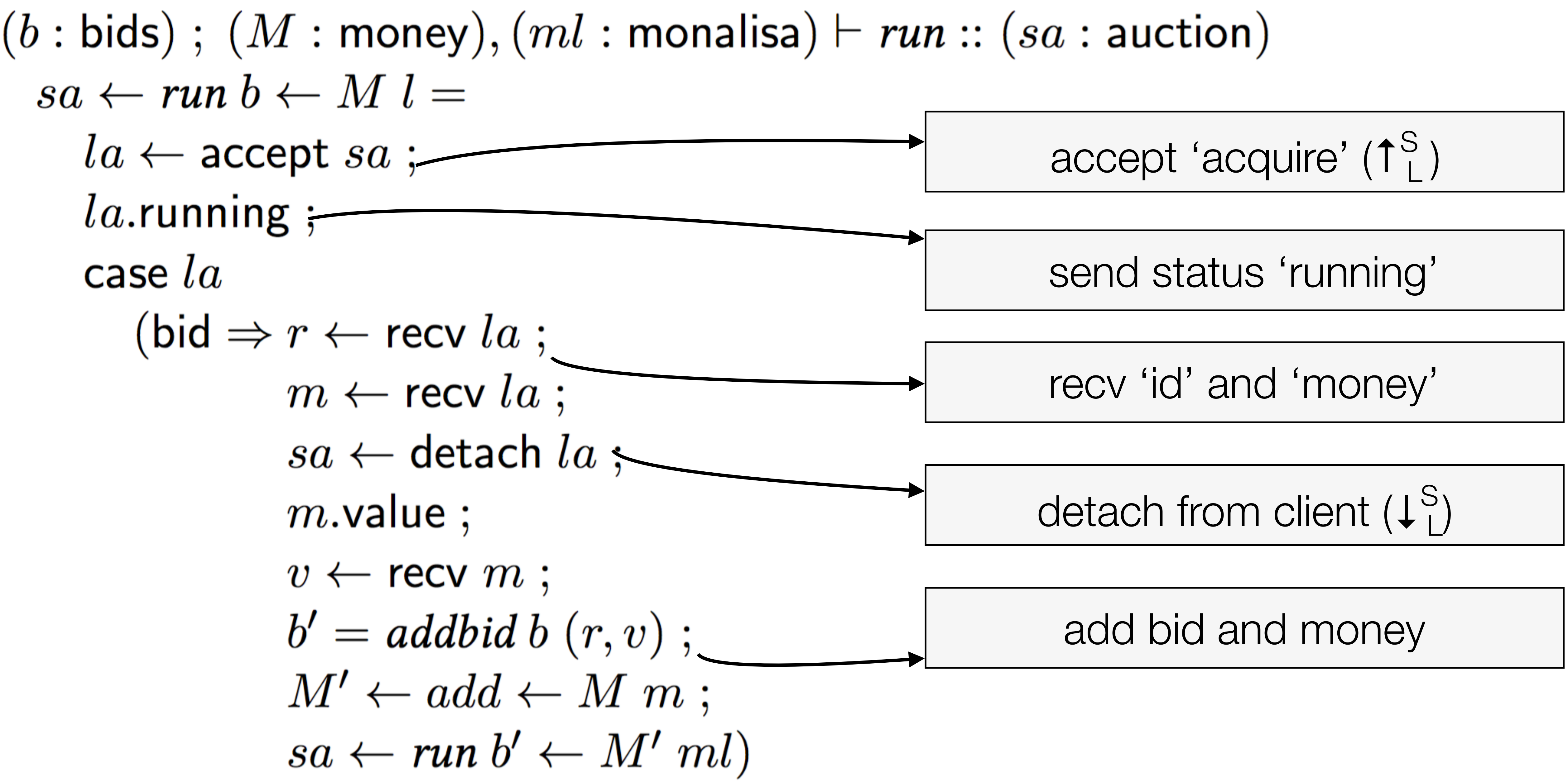
send status ‘running’

recv ‘id’ and ‘money’

detach from client ($\downarrow^{\mathsf{S}}_{\mathsf{L}}$)

Implementation of a Running Auction

$$\mathsf{auction} = \underline{\uparrow_{\mathsf{L}}^{\mathsf{S}}} \triangleleft^{11} \oplus \{\underline{\mathsf{running}} : \&\{\mathsf{bid} : \underline{\mathsf{id}} \supset \underline{\mathsf{money}} \multimap \triangleright^{1} \underline{\downarrow_{\mathsf{L}}^{\mathsf{S}}} \mathsf{auction}\}\},$$



$$\begin{array}{l}
(b : \mathsf{bids}) \; ; \; (M : \mathsf{money}), (ml : \mathsf{monalisa}) \vdash \mathit{run} :: (sa : \mathsf{auction}) \\
\quad sa \leftarrow \mathit{run}\; b \leftarrow M\; l = \\
\qquad la \leftarrow \mathsf{accept}\; sa \; ; \\
\qquad la.\mathsf{running} \; ; \\
\qquad \mathsf{case}\; la \\
\qquad\quad (\mathsf{bid} \Rightarrow r \leftarrow \mathsf{recv}\; la \; ; \\
\qquad\qquad\quad m \leftarrow \mathsf{recv}\; la \; ; \\
\qquad\qquad\quad sa \leftarrow \mathsf{detach}\; la \; ; \\
\qquad\qquad\quad m.\mathsf{value} \; ; \\
\qquad\qquad\quad v \leftarrow \mathsf{recv}\; m \; ; \\
\qquad\qquad\quad b' = \mathit{addbid}\; b\; (r, v) \; ; \\
\qquad\qquad\quad M' \leftarrow \mathit{add} \leftarrow M\; m \; ; \\
\qquad\qquad\quad sa \leftarrow \mathit{run}\; b' \leftarrow M'\; ml)
\end{array}$$

accept ‘acquire’ ($\uparrow_{\mathsf{L}}^{\mathsf{S}}$)

send status ‘running’

recv ‘id’ and ‘money’

detach from client ($\downarrow_{\mathsf{L}}^{\mathsf{S}}$)

add bid and money

Implementation of a Running Auction

$$\mathsf{auction} = \underline{\uparrow^{\mathsf{S}}_{\mathsf{L}}}\triangleleft^{11} \oplus \{\underline{\mathsf{running}} : \&\{\mathsf{bid} : \underline{\mathsf{id}} \supset \underline{\mathsf{money}} \multimap \triangleright^{1} \underline{\downarrow^{\mathsf{S}}_{\mathsf{L}}}\mathsf{auction}\}\},$$

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

```
$sa \leftarrow run\ b \leftarrow M\ l =$
    $la \leftarrow$ accept $sa$ ;
    $la$.running ;
    case $la$
      (bid $\Rightarrow r \leftarrow$ recv $la$ ;
                $m \leftarrow$ recv $la$ ;
                $sa \leftarrow$ detach $la$ ;
                $m$.value ;
                $v \leftarrow$ recv $m$ ;
                $b' = addbid\ b\ (r, v)$ ;
                $M' \leftarrow add \leftarrow M\ m$ ;
                $sa \leftarrow run\ b' \leftarrow M'\ ml$)
```

accept ‘acquire’ ($\uparrow^{\mathsf{S}}_{\mathsf{L}}$)

send status ‘running’

recv ‘id’ and ‘money’

detach from client ($\downarrow^{\mathsf{S}}_{\mathsf{L}}$)

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

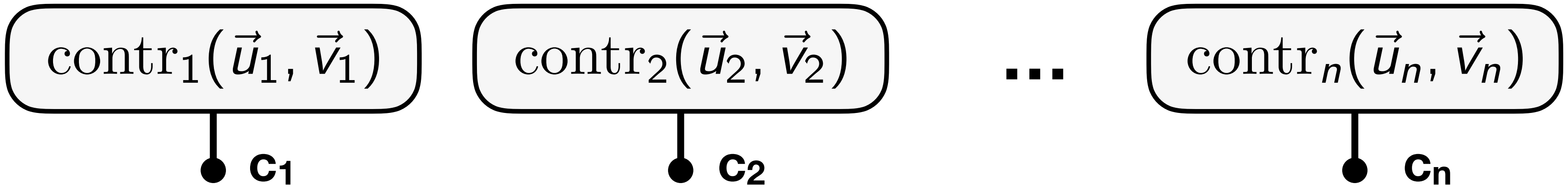
- Add an additional argument that carries potential
- User arg N ~ maximal number of players => gas bound is $81*N + 28$

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



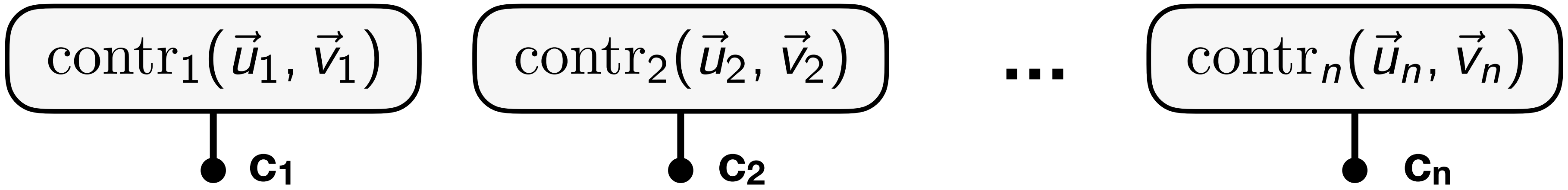
$\mathrm{contr}_1(\vec{u}_1, \vec{v}_1)$ $\mathrm{contr}_2(\vec{u}_2, \vec{v}_2)$ … $\mathrm{contr}_n(\vec{u}_n, \vec{v}_n)$

c_1 **c_2** **c_n**

Computation on a Blockchain

Blockchain state: shared processes waiting to be acquired

Contracts store functional and linear data.



$\text{contr}_1(\vec{u}_1, \vec{v}_1)$ $\mathbf{c_1}$

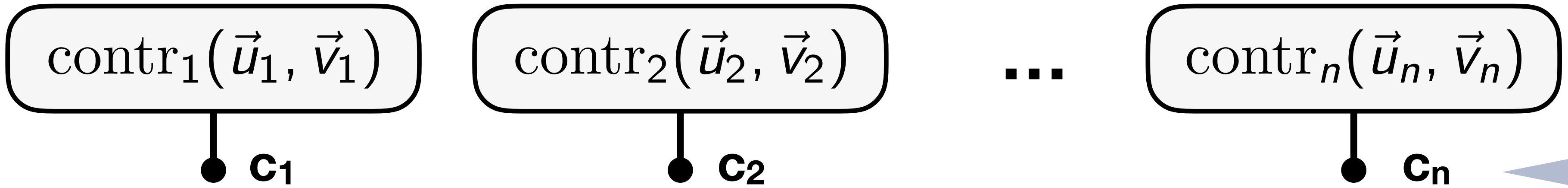
$\text{contr}_2(\vec{u}_2, \vec{v}_2)$ $\mathbf{c_2}$

...

$\text{contr}_n(\vec{u}_n, \vec{v}_n)$ $\mathbf{c_n}$

Computation on a Blockchain

Blockchain state: shared processes waiting to be acquired



$\mathrm{contr}_1(\vec{u}_1, \vec{v}_1)$ $\mathrm{contr}_2(\vec{u}_2, \vec{v}_2)$ … $\mathrm{contr}_n(\vec{u}_n, \vec{v}_n)$

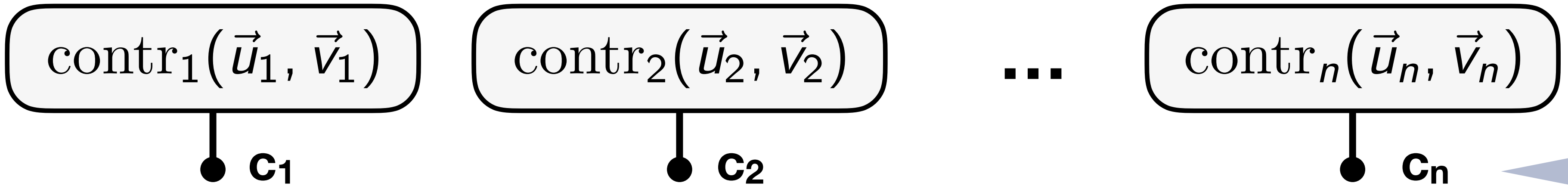
Contracts store functional and linear data.

Channel name = address

Computation on a Blockchain

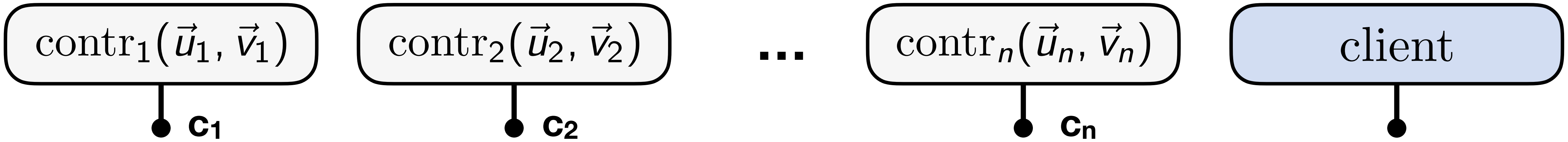
Blockchain state: shared processes waiting to be acquired

Contracts store functional and linear data.



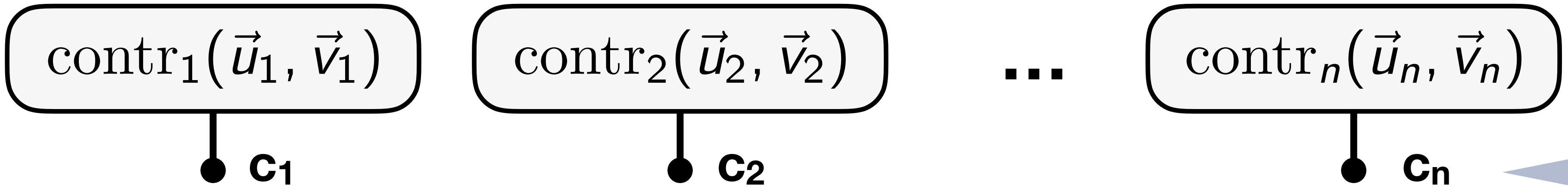
Channel name = address

Transaction: client submits code of a linear process



Computation on a Blockchain

Blockchain state: shared processes waiting to be acquired



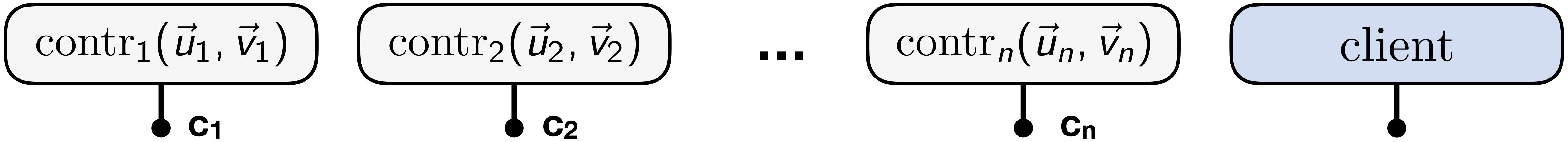
$\mathrm{contr}_1(\vec{u}_1, \vec{v}_1)$ $\mathrm{contr}_2(\vec{u}_2, \vec{v}_2)$ … $\mathrm{contr}_n(\vec{u}_n, \vec{v}_n)$

c_1 c_2 c_n

Contracts store functional and linear data.

Channel name = address

Transaction: client submits code of a linear process



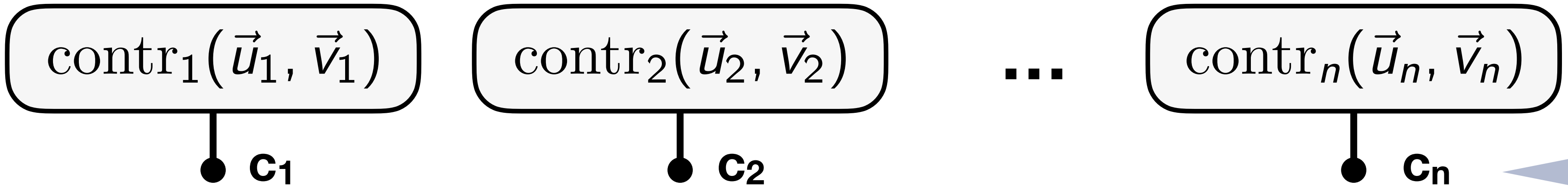
$\mathrm{contr}_1(\vec{u}_1, \vec{v}_1)$ $\mathrm{contr}_2(\vec{u}_2, \vec{v}_2)$ … $\mathrm{contr}_n(\vec{u}_n, \vec{v}_n)$ client

c_1 c_2 c_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

Computation on a Blockchain

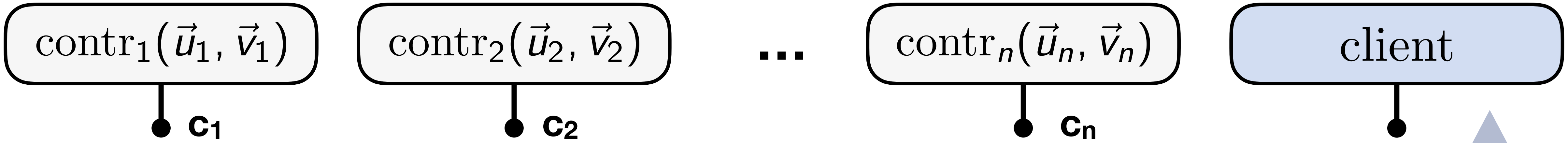
Blockchain state: shared processes waiting to be acquired



Contracts store functional and linear data.

Channel name = address

Transaction: client submits code of a linear process



Contract should have default clients.

- 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

Blockchain, Type Checking, and Verification

Type checking is part of the attack surface

- Contract code can 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 guaranties 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)
- LICS ’18: Resource-Aware Session Types
- arXiv ’19: Nomos

Ongoing work: implementation

- Parser ✓
- Type checker ✓
- Interpreter
- Compiler

Nomos

Collaborators: Stephanie Balzer, Ankush Das, and Frank Pfenning

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