Exact and Linear-Time Gas-Cost Analysis

Ankush Das* (Carnegie Mellon University)
Shaz Qadeer (Facebook)

SAS 2020

What are Smart Contracts?

- Programs to digitally facilitate the execution of a transaction between distrusting parties
- Transactions are executed by third-party miners and stored on a global distributed ledger, or blockchain
- User pays for the execution cost of transaction in gas units

What are Smart Contracts?

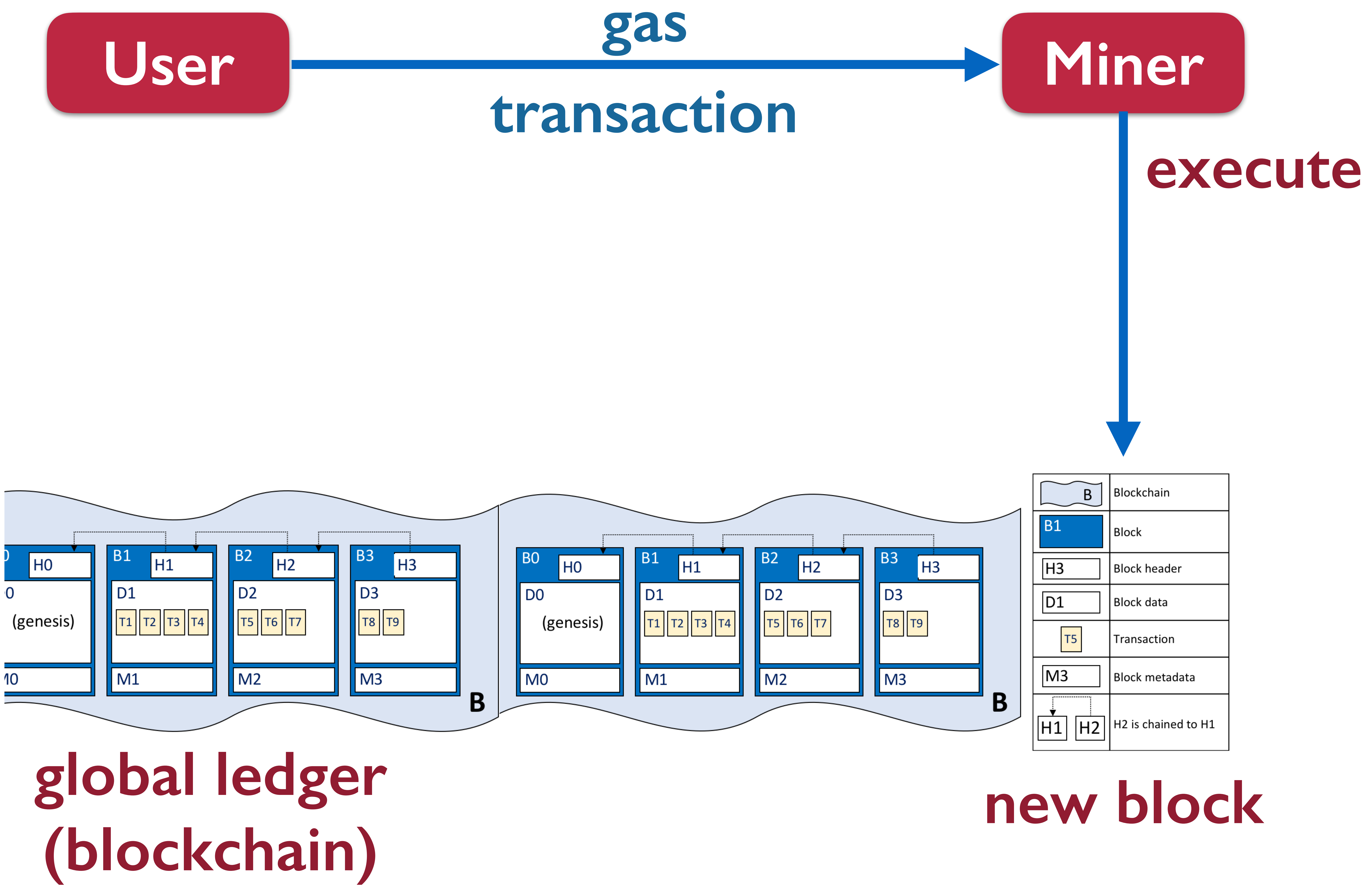
- Programs to digitally facilitate the execution of a transaction between distrusting parties
- Transactions are executed by third-party miners and stored on a global distributed ledger, or blockchain
- User pays for the execution cost of transaction in gas units

set standard assigns cost to each operation

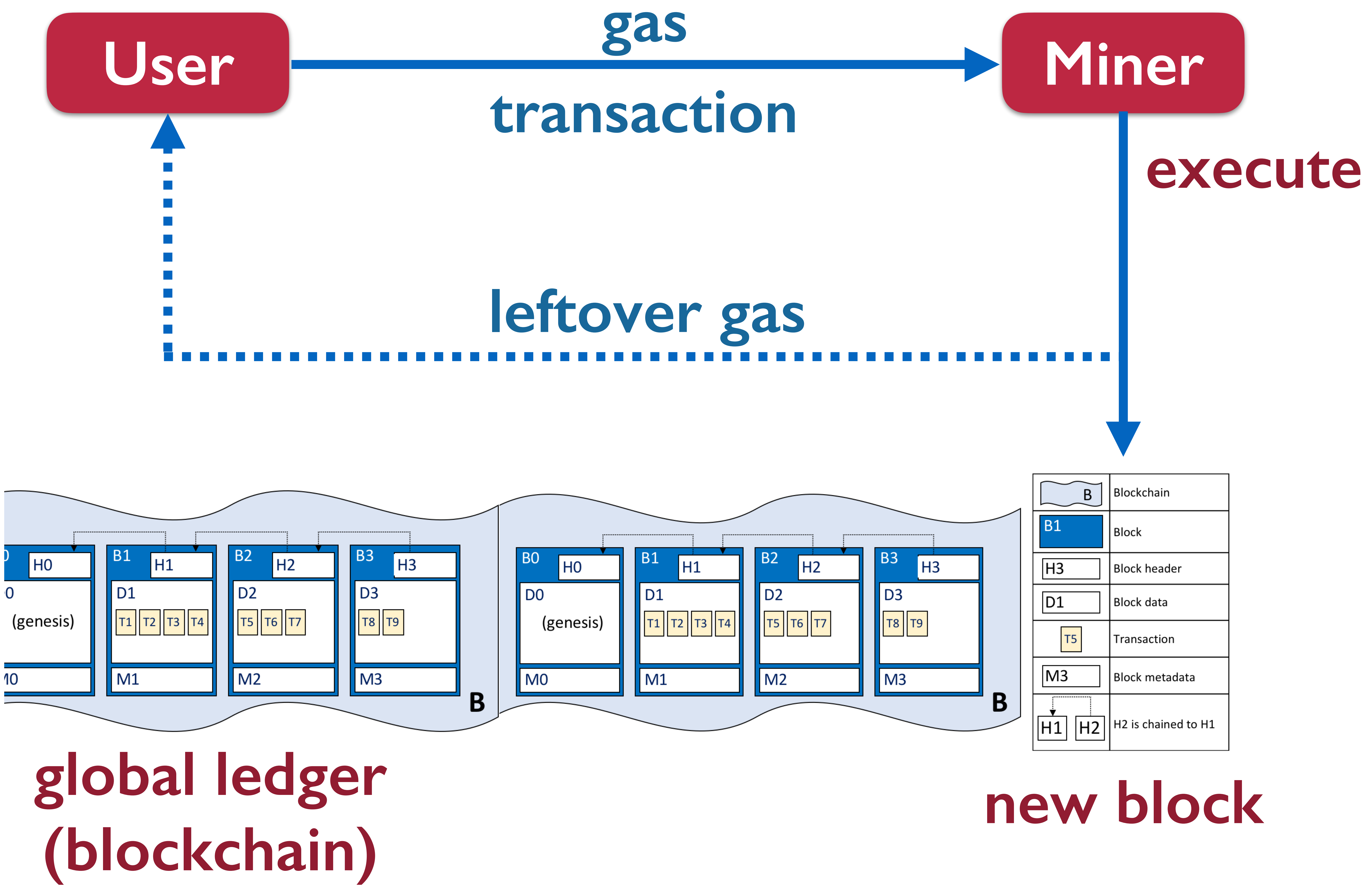
Execution Model

Execution Model

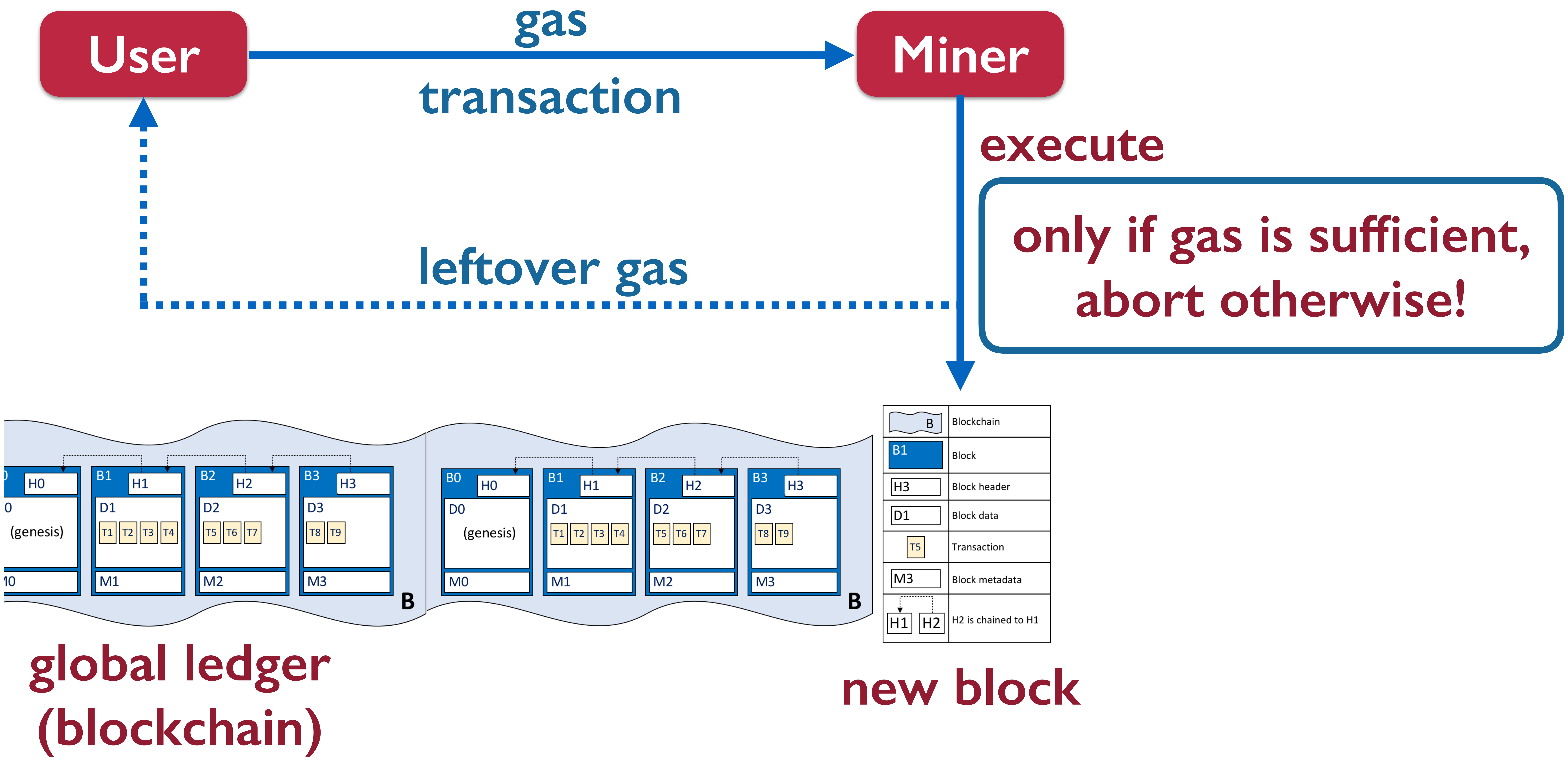
Execution Model



Execution Model



Execution Model



Prior Work on Gas Analysis

- Many tools for computing upper bound on gas cost statically (Gastap, Gasol, Nomos, etc.)
- Upper gas bounds are inadequate
  - Blockchains still track gas dynamically to return leftover gas, *creates runtime overhead*
  - Miners fit transactions in a block based on *exact gas cost*
- Need for *exact gas-cost analysis*
- Analysis should be *efficient*, otherwise can cause *DoS attacks*

Our Work — GasBoX

- GasBoX tool computes *exact* gas bound *statically* and *automatically* (relies on LP solver)
- Infers *constant* gas bounds in *linear-time*
- *Eliminates dynamic gas tracking*, compensates runtime overhead
- *Hoare-logic* style reasoning with an abstract notion of gas tank

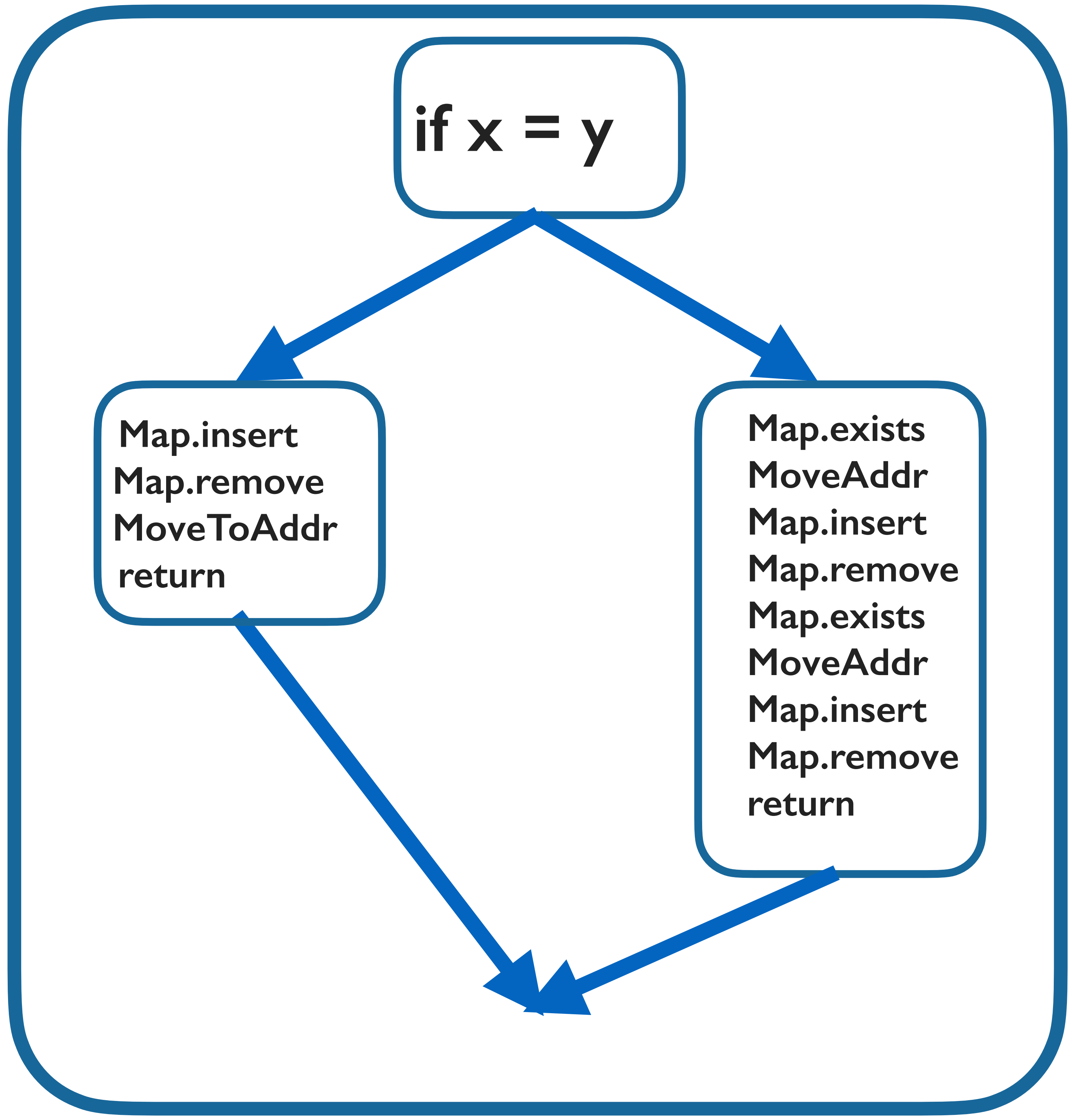
Our Work — GasBoX

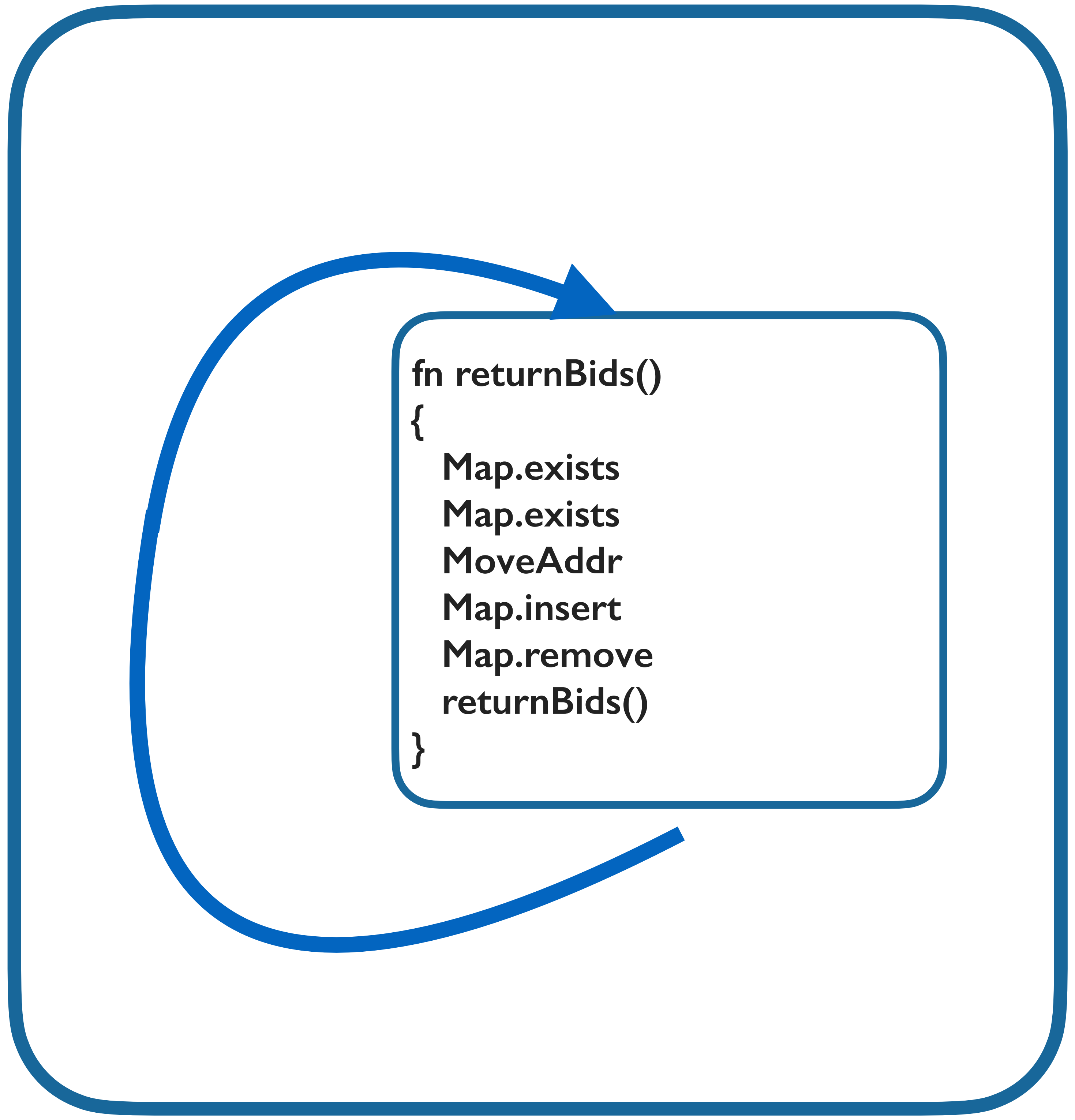
- GasBoX tool computes *exact* gas bound *statically* and *automatically* (relies on LP solver)
- Infers *constant* gas bounds in *linear-time*
- *Eliminates dynamic gas tracking*, compensates runtime overhead
- *Hoare-logic* style reasoning with an abstract notion of gas tank

$$\{tank = \phi + \mathcal{C}(e)\}\ e\ \{tank = \phi \mid \phi \geq 0\}$$

$\mathcal{C}$(e): cost of executing expression e

Challenges





Exact gas bound in the presence of branches

Constant gas bound in the presence of recursion

Bidding in an Auction


```
fn addBid(bidmap: &Map<addr, Coin>, bidder: addr, bid: Coin)
{
  if Map.exists(bidmap, bidder)
  then
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
  else
    tick(CMapInsert);
    Map.insert(bidmap, bidder, bid);
}
```

tick simulates a cost model

Bidding in an Auction


```
fn addBid(bidmap: &Map<addr, Coin>, bidder: addr, bid: Coin)
{
  if Map.exists(bidmap, bidder)
  then
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
  else
    tick(CMapInsert);
    Map.insert(bidmap, bidder, bid);
}
```

tick simulates a cost model

cost of then branch:
CMoveToAddr

cost of else branch:
CMapInsert

Bidding in an Auction


```
fn addBid(bidmap: &Map<addr, Coin>, bidder: addr, bid: Coin)
{
  if Map.exists(bidmap, bidder)
  then
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
  else
    tick(CMapInsert);
    Map.insert(bidmap, bidder, bid);
}
```

tick simulates a cost model

cost of then branch: CMoveToAddr

≠

cost of else branch: CMapInsert

Cannot determine exact gas cost statically!

Gas.deposit operation


```
fn addBid(bidmap: &Map<addr, Coin>, bidder: addr, bid: Coin)
{
  if Map.exists(bidmap, bidder)
  then
    tick(CMoveToAddr);
    Gas.deposit(CMapInsert);
    MoveToAddr(bidder, bid);
  else
    tick(CMapInsert);
    Gas.deposit(CMoveToAddr);
    Map.insert(bidmap, bidder, bid);
}
```

deposits the corresponding gas units in sender' account

Gas.deposit operation


```
fn addBid(bidmap: &Map<addr, Coin>, bidder: addr, bid: Coin)
{
  if Map.exists(bidmap, bidder)
  then
    tick(CMoveToAddr);
    Gas.deposit(CMapInsert);
    MoveToAddr(bidder, bid);
  else
    tick(CMapInsert);
    Gas.deposit(CMoveToAddr);
    Map.insert(bidmap, bidder, bid);
}
```

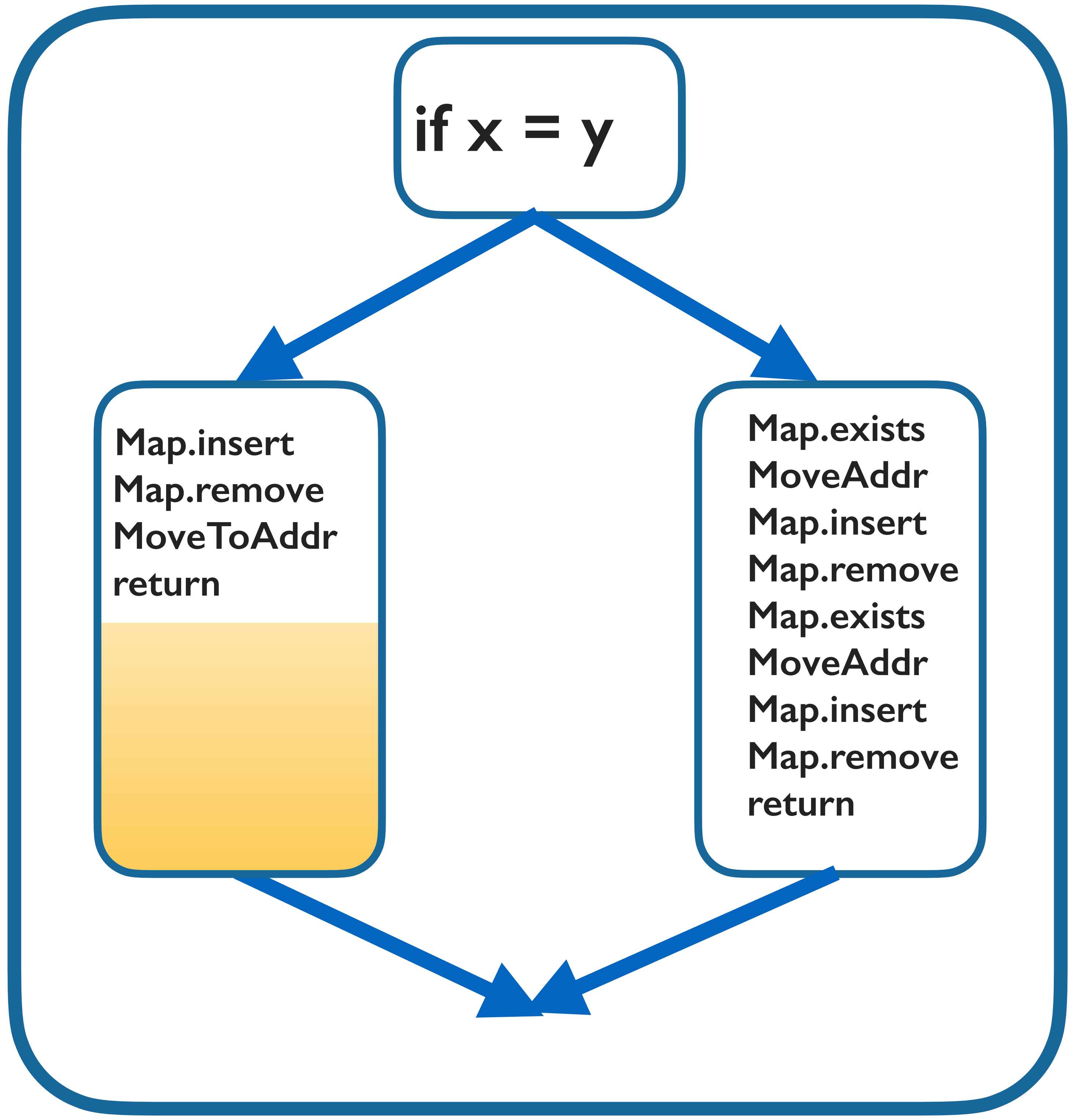
deposits the corresponding gas units in sender' account

cost of then branch: CMoveToAddr + CMapInsert

=

cost of else branch: CMapInsert + CMoveToAddr

Advantages of Gas.deposit



Less costly branch augmented with Gas.deposit

- Returns leftover gas to sender of transaction
- *Eliminates* dynamic gas metering
- GasBox inserts deposit operations *automatically*
- *No burden* on the programmer

Handling Unbounded Computation


```
fn returnBids(bidmap : &Map<addr, Coin>)
{
  if Map.size(bidmap) > 0
  then
    let (bidder, bid) = Map.remove_first(bidmap);
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
    returnBids(bidmap);
}
```

Handling Unbounded Computation


```
fn returnBids(bidmap : &Map<addr, Coin>)
{
  if Map.size(bidmap) > 0
  then
    let (bidder, bid) = Map.remove_first(bidmap);
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
    returnBids(bidmap);
}
```

cost of returnBids =

CMoveToAddr * *sizeof* (bidmap)

Gas cost is not a constant, depends on argument size!

Gas Amortization


```
resource GasBid
{
  gas : Gas(CMoveToAddr),
  bid : Coin
}

fn addBid(bidmap: &Map<addr, GasBid>, bidder: addr, b: Coin)
{
  if Map.exists(bidmap, bidder)
  then
    tick(CMoveToAddr);
    MoveToAddr(bidder, b);
    Gas.deposit(CMapInsert + CMoveToAddr);
  else
    let g = Gas.construct(CMoveToAddr);
    let gbid = pack<GasBid> {gas: g, bid: b};
    tick(CMapInsert);
    Map.insert(bidmap, bidder, gbid);
    Gas.deposit(CMoveToAddr);
}
```

Gas Amortization


```
resource GasBid
{
  gas : Gas(CMoveToAddr),
  bid : Coin
}

fn addBid(bidmap: &Map<addr, GasBid>, bidder: addr, b: Coin)
{
  if Map.exists(bidmap, bidder)
  then
    tick(CMoveToAddr);
    MoveToAddr(bidder, b);
    Gas.deposit(CMapInsert + CMoveToAddr);
  else
    let g = Gas.construct(CMoveToAddr);
    let gbid = pack<GasBid> {gas: g, bid: b};
    tick(CMapInsert);
    Map.insert(bidmap, bidder, gbid);
    Gas.deposit(CMoveToAddr);
}
```

store CMoveToAddr gas units inside a bid

Gas Amortization


```
resource GasBid
{
  gas : Gas(CMoveToAddr),
  bid : Coin
}

fn addBid(bidmap: &Map<addr, GasBid>, bidder: addr, b: Coin)
{
  if Map.exists(bidmap, bidder)
  then
    tick(CMoveToAddr);
    MoveToAddr(bidder, b);
    Gas.deposit(CMapInsert + CMoveToAddr);
  else
    let g = Gas.construct(CMoveToAddr);
    let gbid = pack<GasBid> {gas: g, bid: b};
    tick(CMapInsert);
    Map.insert(bidmap, bidder, gbid);
    Gas.deposit(CMoveToAddr);
}
```

store CMoveToAddr gas units inside a bid

cost of addBid =

2 * CMoveToAddr + CMapInsert

Stored Gas pays for Recursion


```
fn returnBids(bidmap : &Map<addr, GasBid>)
{
  if (Map.size(bidmap) > 0)
  then
    let (bidder, gbid) = Map.remove_first(bidmap);
    let (g, bid) = unpack<GasBid>(gbid);
    Gas.destruct(g);
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
    returnBids(bidmap);
}
```

Stored Gas pays for Recursion


```
fn returnBids(bidmap : &Map<addr, GasBid>)
{
  if (Map.size(bidmap) > 0)
  then
    let (bidder, gbid) = Map.remove_first(bidmap);
    let (g, bid) = unpack<GasBid>(gbid);
    Gas.destruct(g);
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
    returnBids(bidmap);
}
```

unpack gbid to release the gas inside it

Stored Gas pays for Recursion


```
fn returnBids(bidmap : &Map<addr, GasBid>)
{
  if (Map.size(bidmap) > 0)
  then
    let (bidder, gbid) = Map.remove_first(bidmap);
    let (g, bid) = unpack<GasBid>(gbid);
    Gas.destruct(g);
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
    returnBids(bidmap);
}
```

unpack gbid to release the gas inside it

destruct the gas to pay for ticks

Stored Gas pays for Recursion

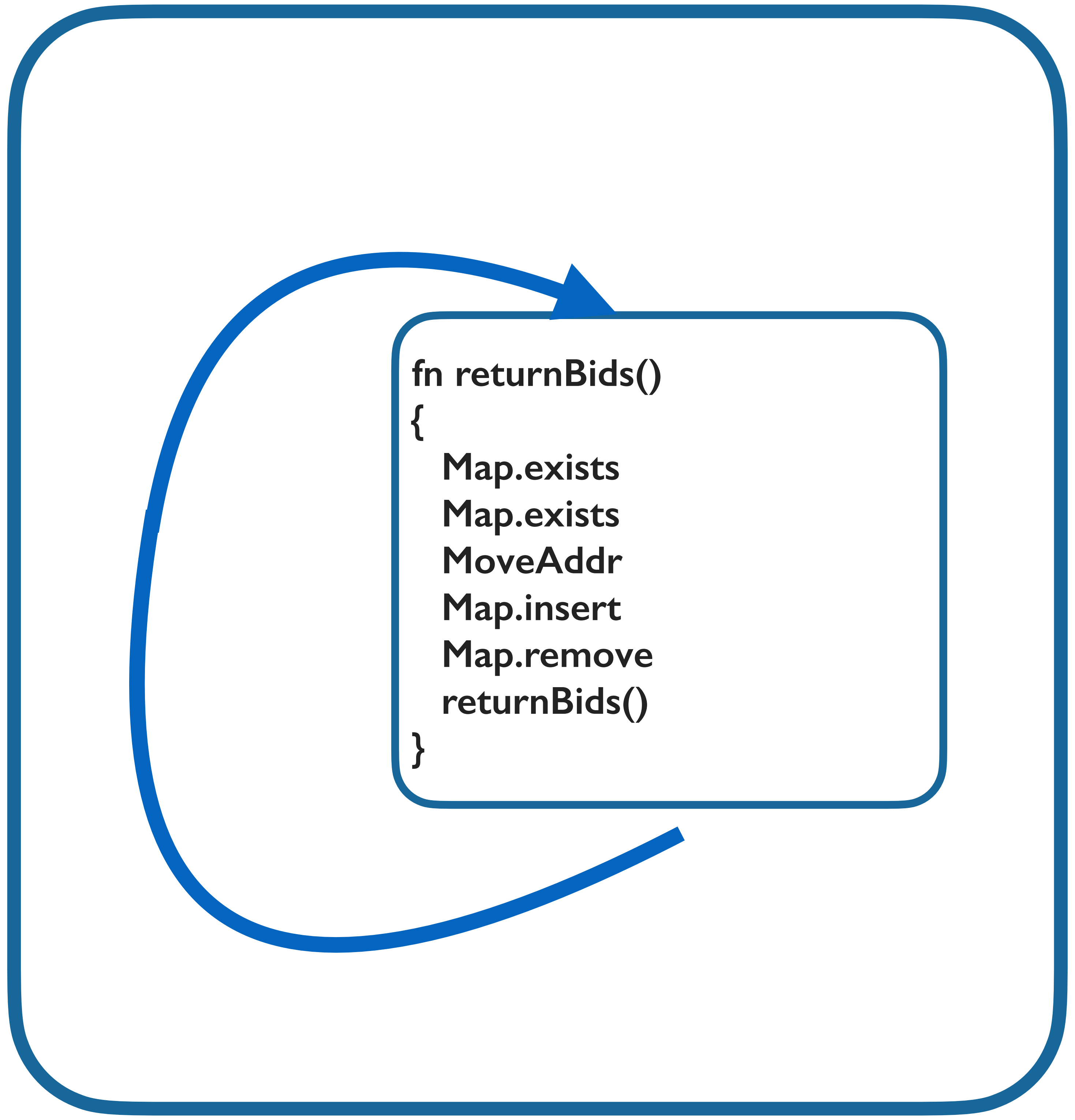

```
fn returnBids(bidmap : &Map<addr, GasBid>)
{
  if (Map.size(bidmap) > 0)
  then
    let (bidder, gbid) = Map.remove_first(bidmap);
    let (g, bid) = unpack<GasBid>(gbid);
    Gas.destruct(g);
    tick(CMoveToAddr);
    MoveToAddr(bidder, bid);
    returnBids(bidmap);
}
```

unpack gbid to release the gas inside it

destruct the gas to pay for ticks

cost of returnBids = 0

Advantages of Gas Amortization



Gas amortization pays for the cost of recursion

- *Simplifies* analysis (no need to track sizes of data structures)
- Gas bound of returnBids *cannot exceed block gas limit*
- *Equitable gas-distribution:* bidder pays for the gas cost of return of their bid
- *Constant gas bound* even though unbounded gas cost!

Inference of Gas Bounds


```
resource GasBid
{
  gas : Gas(*),
  bid : Coin
}
```

```
resource GasBid
{
  gas : Gas(CMoveToAddr),
  bid : Coin
}
```

- GasBoX *infers gas bounds of all functions* in the program
- Also infers the gas to be *stored inside data structures*
- Programmers only need to *indicate where gas needs to be stored*
- Low programmer burden!

Inference of Gas Bounds

```
resource GasBid
{
  gas : Gas(*),
  bid : Coin
}
```

```
resource GasBid
{
  gas : Gas(CMoveToAddr),
  bid : Coin
}
```

- GasBoX *infers gas bounds of all functions* in the program
- Also infers the gas to be *stored inside data structures*
- Programmers only need to *indicate where gas needs to be stored*
- Low programmer burden!

Formal Analysis

$$\{tank = \phi + \mathcal{C}(e)\}\ e\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi + n\}\ \texttt{tick}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi + n\}\ \texttt{Gas.deposit}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi + n\}\ \texttt{Gas.construct}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi \mid v : \texttt{Gas}(n)\}\ \texttt{Gas.destruct}(v)\ \{tank = \phi + n\}$$

$$\{tank = \phi + \mathcal{C}(e)\}\ \texttt{return}\ e\ \{tank = \phi \mid \phi = 0\}$$

Formal Analysis

$$\{tank = \phi + \mathcal{C}(e)\}\ e\ \{tank = \phi \mid \phi \geq 0\}$$

tick, Gas.deposit and Gas.construct remove n units from tank

$$\{tank = \phi + n\}\ \texttt{tick}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi + n\}\ \texttt{Gas.deposit}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi + n\}\ \texttt{Gas.construct}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi \mid v : \texttt{Gas}(n)\}\ \texttt{Gas.destruct}(v)\ \{tank = \phi + n\}$$

$$\{tank = \phi + \mathcal{C}(e)\}\ \texttt{return}\ e\ \{tank = \phi \mid \phi = 0\}$$

Formal Analysis

$\{tank = \phi + \mathcal{C}(e)\}\ e\ \{tank = \phi \mid \phi \geq 0\}$

$\{tank = \phi + n\}\ \texttt{tick}(n)\ \{tank = \phi \mid \phi \geq 0\}$

$\{tank = \phi + n\}\ \texttt{Gas.deposit}(n)\ \{tank = \phi \mid \phi \geq 0\}$

$\{tank = \phi + n\}\ \texttt{Gas.construct}(n)\ \{tank = \phi \mid \phi \geq 0\}$

$\{tank = \phi \mid v : \texttt{Gas}(n)\}\ \texttt{Gas.destruct}(v)\ \{tank = \phi + n\}$

$\{tank = \phi + \mathcal{C}(e)\}\ \texttt{return}\ e\ \{tank = \phi \mid \phi = 0\}$

tick, Gas.deposit and Gas.construct remove n units from tank

Gas.destruct adds n units to tank

Formal Analysis

$$\{tank = \phi + \mathcal{C}(e)\}\ e\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi + n\}\ \texttt{tick}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi + n\}\ \texttt{Gas.deposit}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi + n\}\ \texttt{Gas.construct}(n)\ \{tank = \phi \mid \phi \geq 0\}$$

$$\{tank = \phi \mid v : \texttt{Gas}(n)\}\ \texttt{Gas.destruct}(v)\ \{tank = \phi + n\}$$

$$\{tank = \phi + \mathcal{C}(e)\}\ \texttt{return}\ e\ \{tank = \phi \mid \phi = 0\}$$

tick, Gas.deposit and Gas.construct remove n units from tank

Gas.destruct adds n units to tank

post-gas after return must be 0

Soundness Theorem

If $\{tank = \phi\}\ e\ \{tank = \phi'\}$,

and $e \Downarrow^{\mu}_{\mu'} v$,

then $\phi - \phi' = \mu - \mu'$.

Soundness Theorem

If $\{tank = \phi\}\ e\ \{tank = \phi'\}$,

and $e \Downarrow^{\mu}_{\mu'} v$,

then $\phi - \phi' = \mu - \mu'$.

Static Gas Cost

Soundness Theorem

If $\{tank = \phi\}\ e\ \{tank = \phi'\}$,

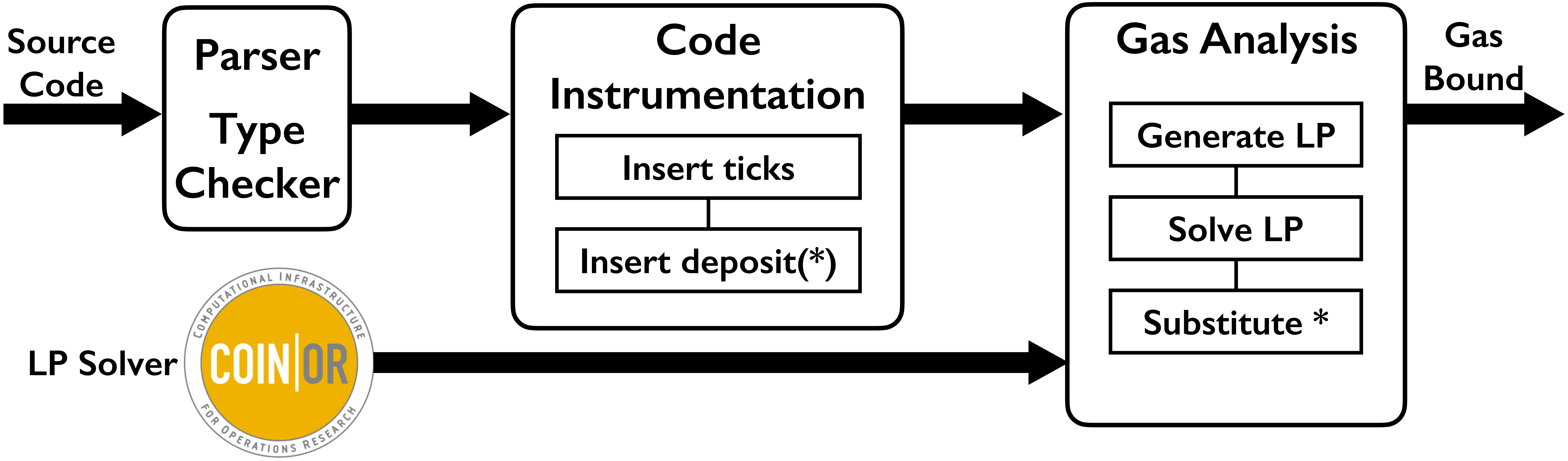
and $e \Downarrow^{\mu}_{\mu'} v$,

then $\phi - \phi' = \mu - \mu'$.

Static Gas Cost

Dynamic Gas Cost

Workflow of GasBoX



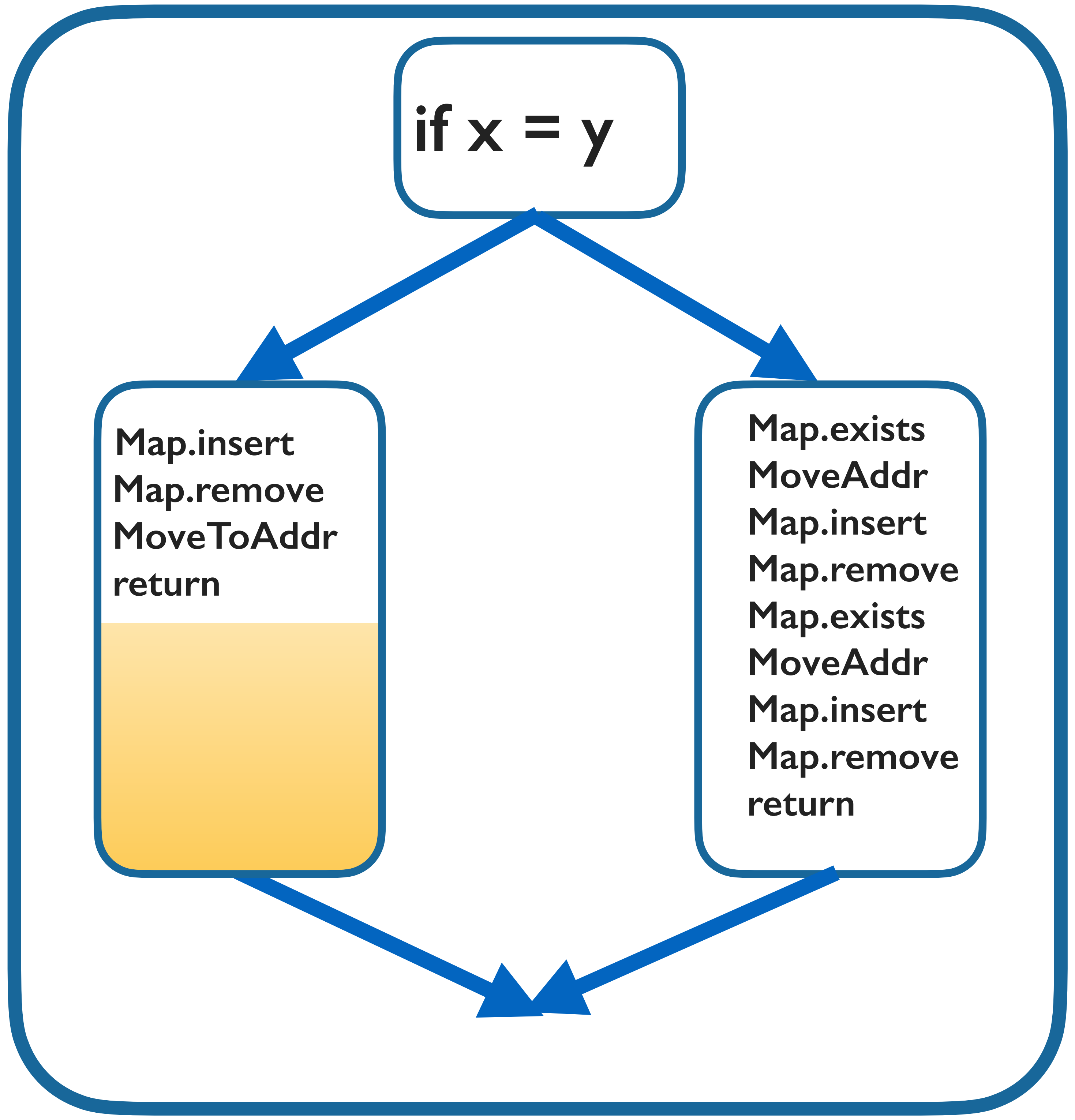
- ticks and deposits are automatically inserted by GasBoX
- relies on type checker and LP solver for inferring bounds

Evaluation

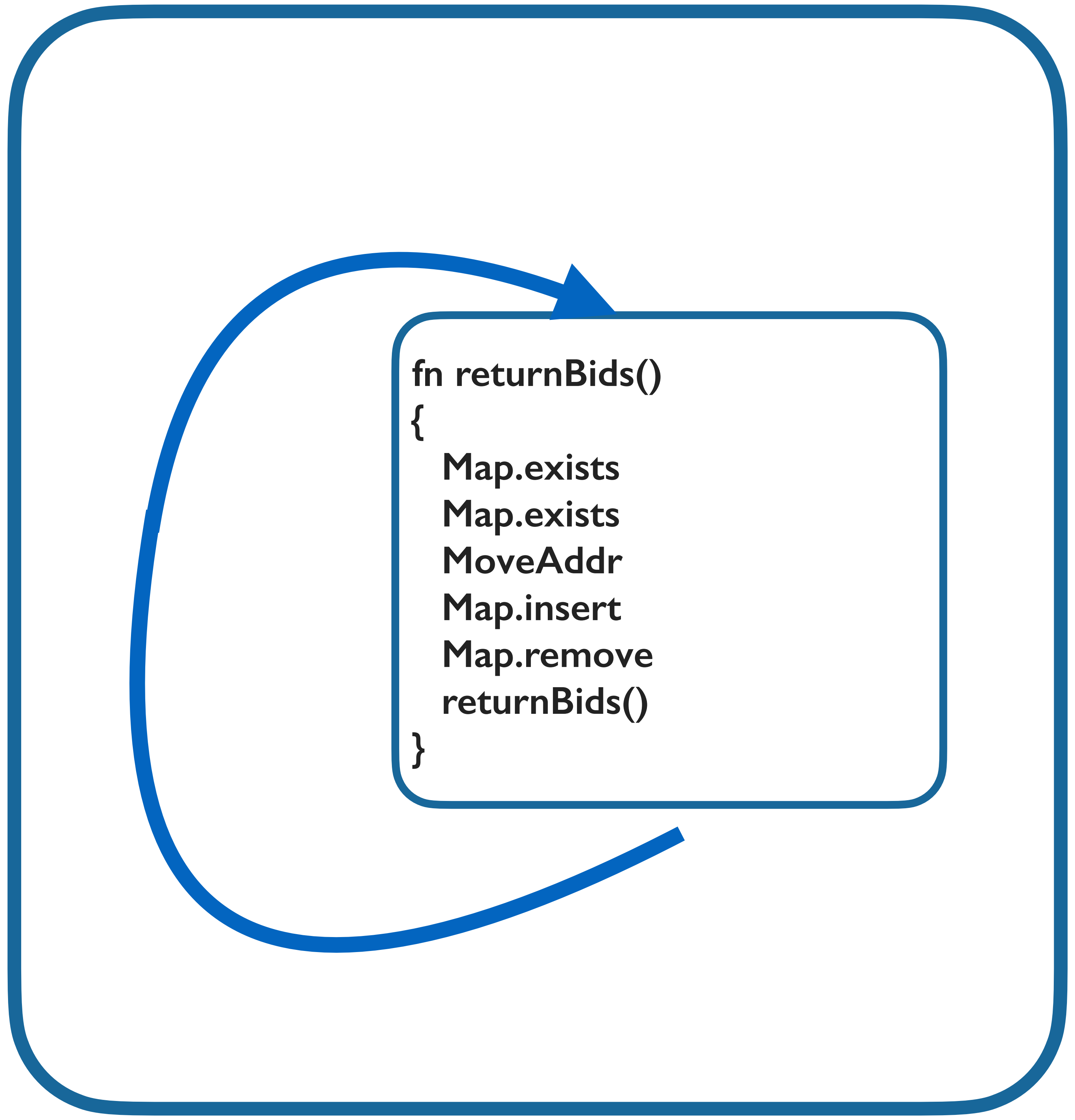
Contract	LOC	Defs	Vars	Cons	I (ms)	V (μs)
auction	44	7	3	44	3.05	12.16
bank	138	14	11	254	3.97	54.12
ERC 20	101	11	8	191	3.45	56.98
escrow	140	7	9	213	3.29	61.03
insurance	43	5	3	43	3.02	9.05
voting	75	7	8	131	3.19	30.99
wallet	74	8	5	158	3.35	52.93
ethereumpot	259	13	13	332	3.94	101.08
puzzle	62	6	6	91	3.13	15.97
amort. auction	70	7	5	62	2.99	15.02
amort. bank	189	17	17	347	4.44	73.19
tether	382	29	30	842	26.14	365.01
libra system	124	12	12	170	3.38	45.06
Total	**1701**	**143**	**130**	**2878**	**67.34**	**892.59**

- LOC: lines of code
- Defs: functions
- Vars, Cons: variables and constraints in LP
- I (ms): inference time in milliseconds
- V(us): verification time in microseconds

Conclusion

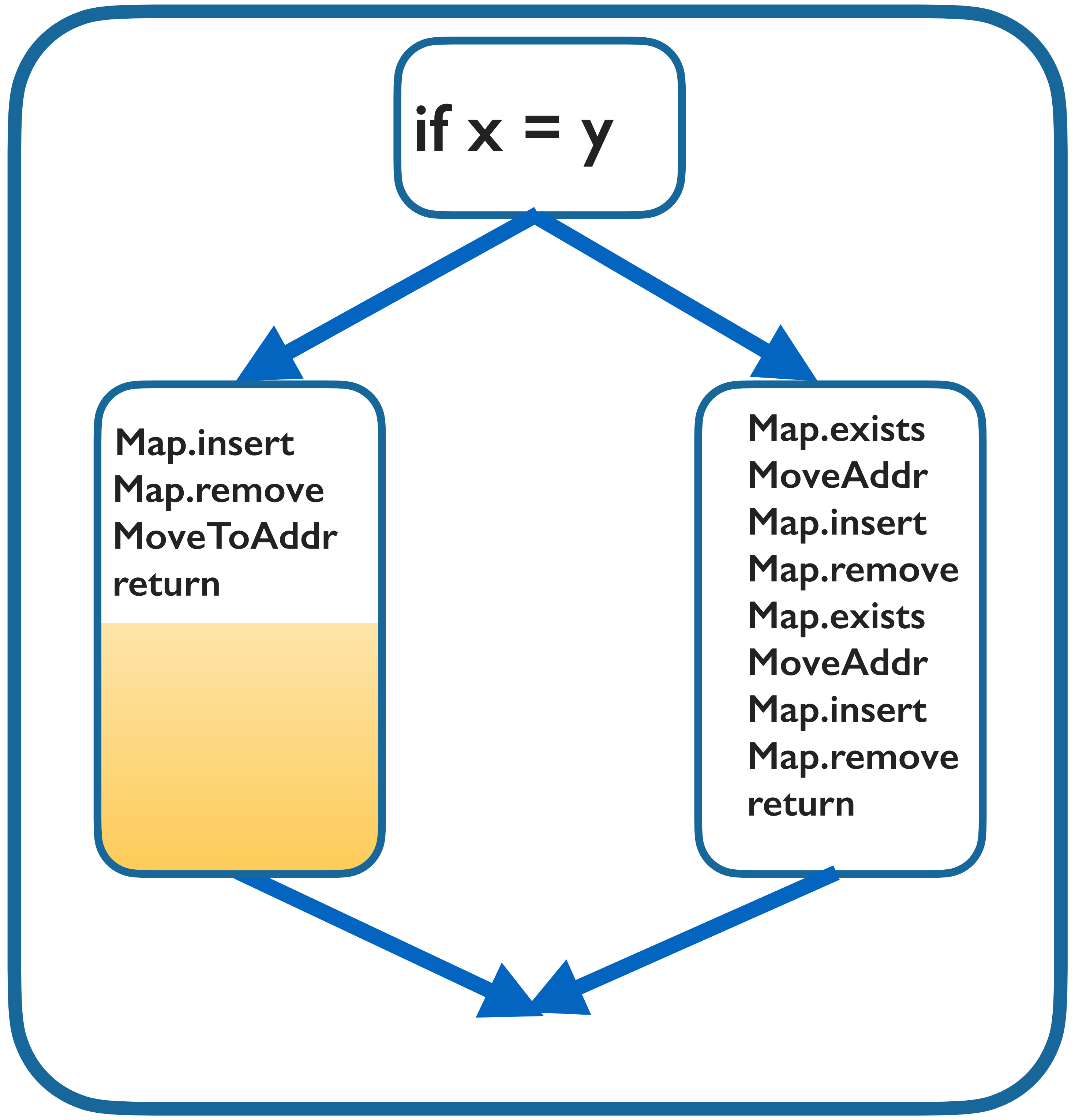


Less costly branch augmented with Gas.deposit

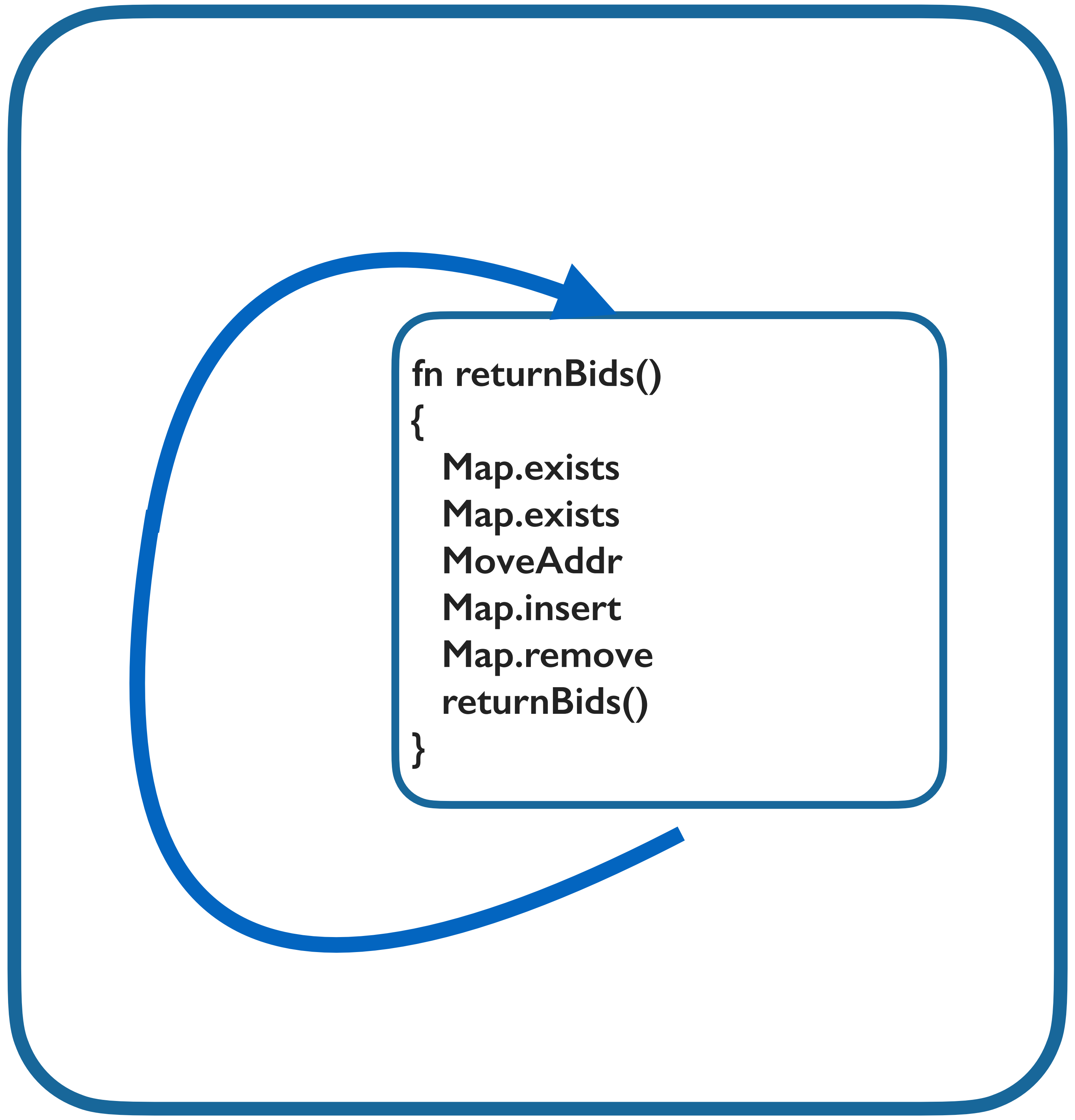


Gas amortization pays for the cost of recursion

Conclusion



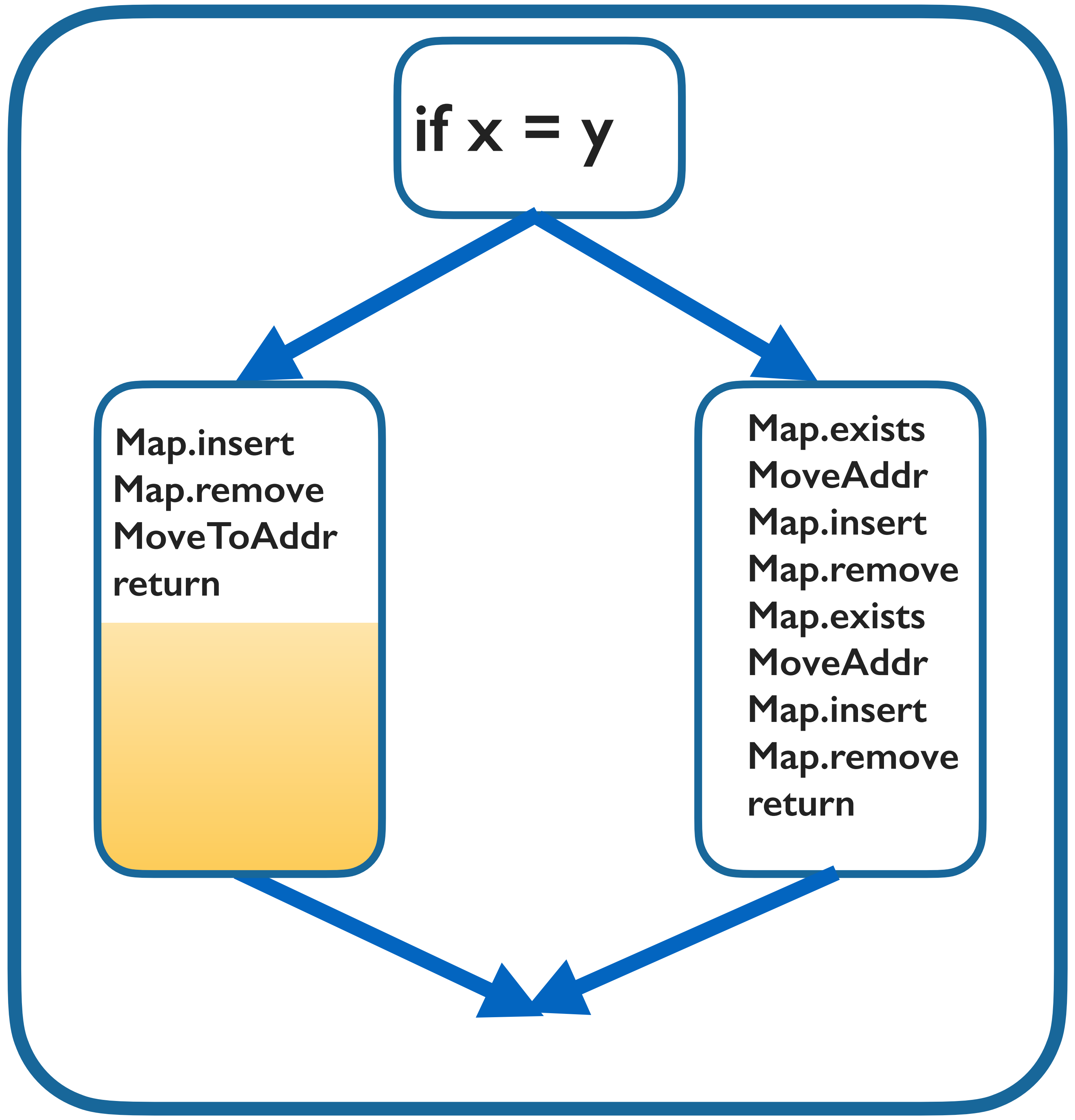
Hoare-logic style gas-analysis framework



Less costly branch augmented with Gas.deposit

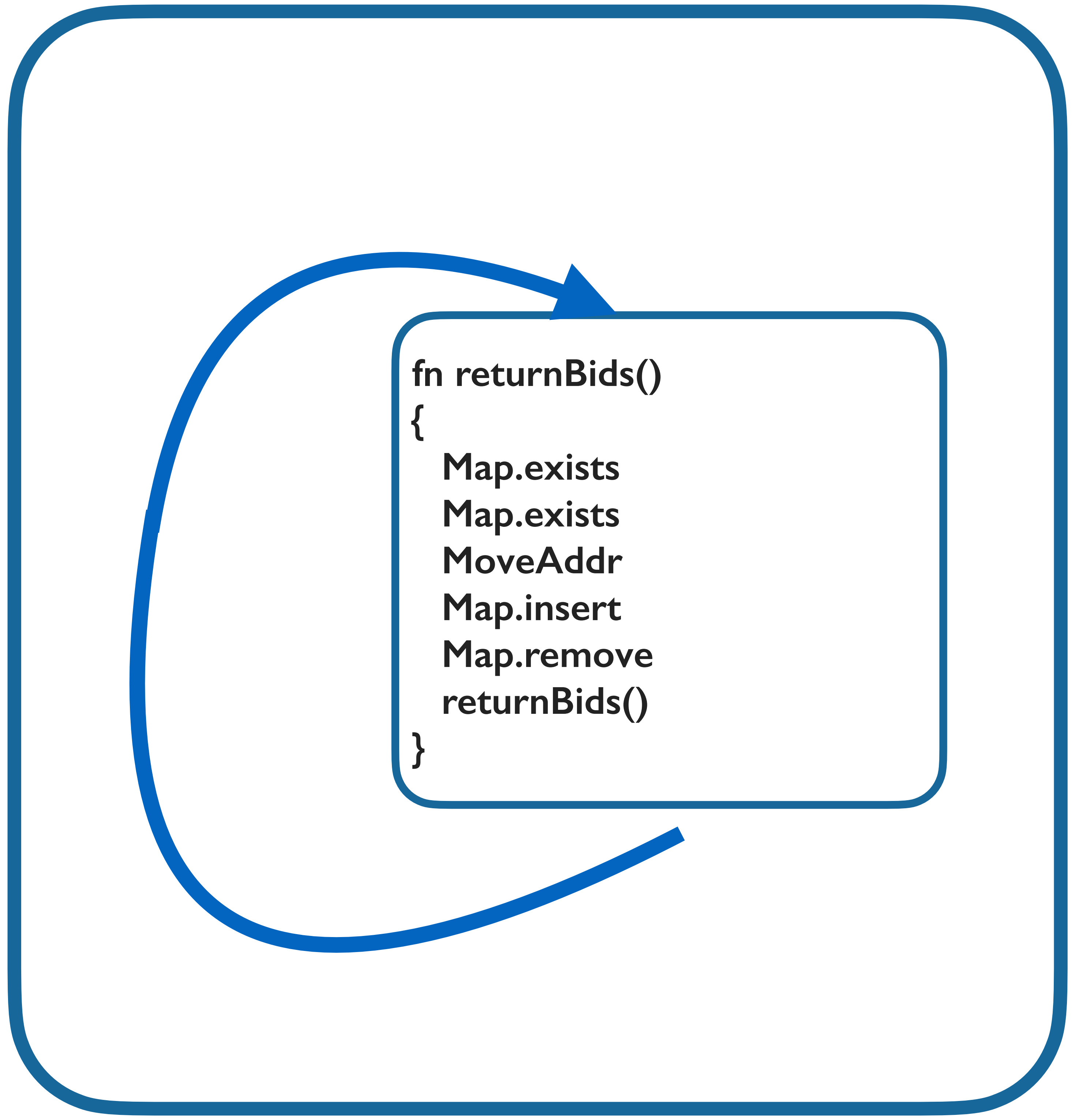
Gas amortization pays for the cost of recursion

Conclusion



Hoare-logic style gas-analysis framework

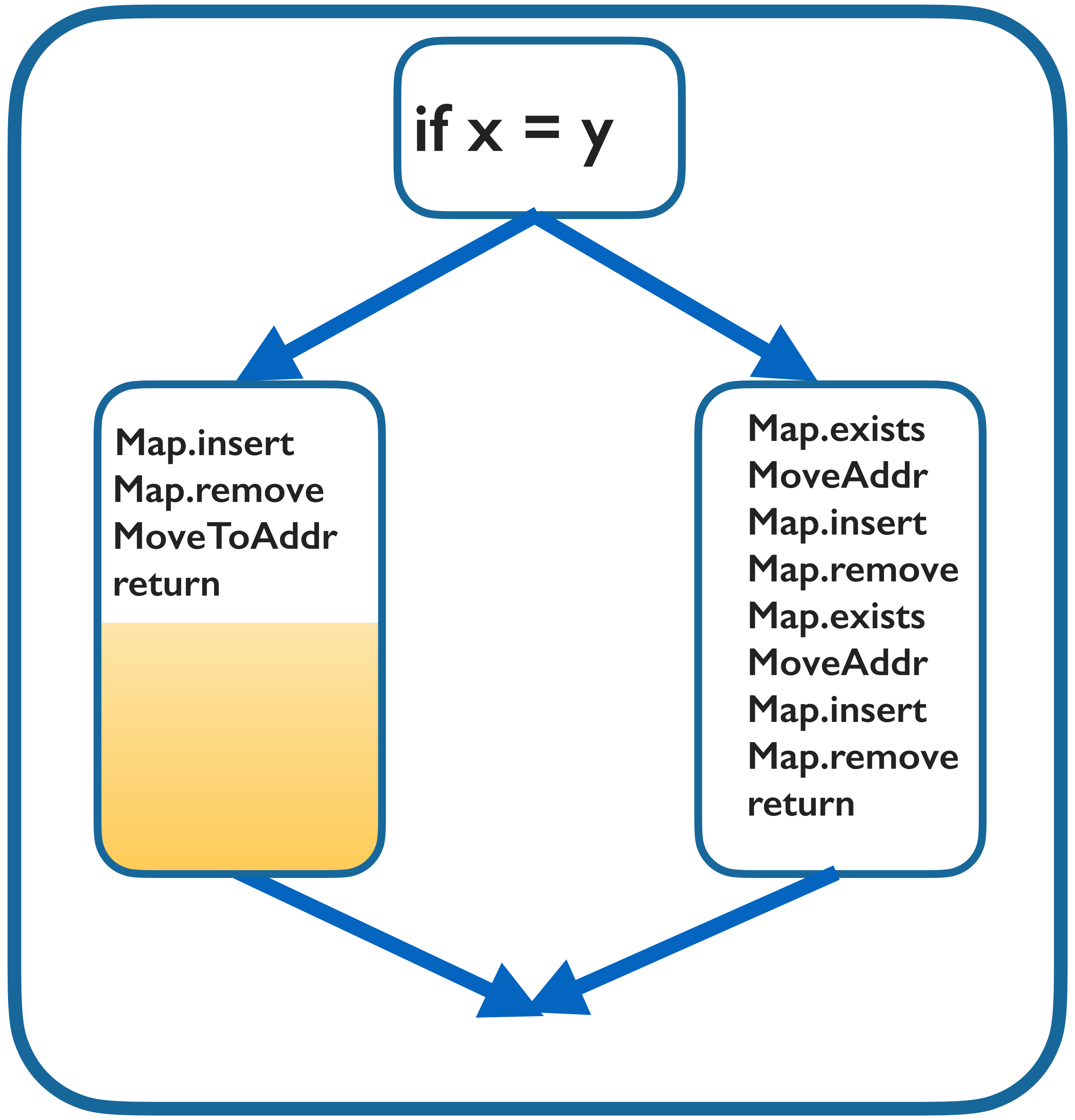
inference of exact gas bounds in linear-time



Less costly branch augmented with Gas.deposit

Gas amortization pays for the cost of recursion

Conclusion

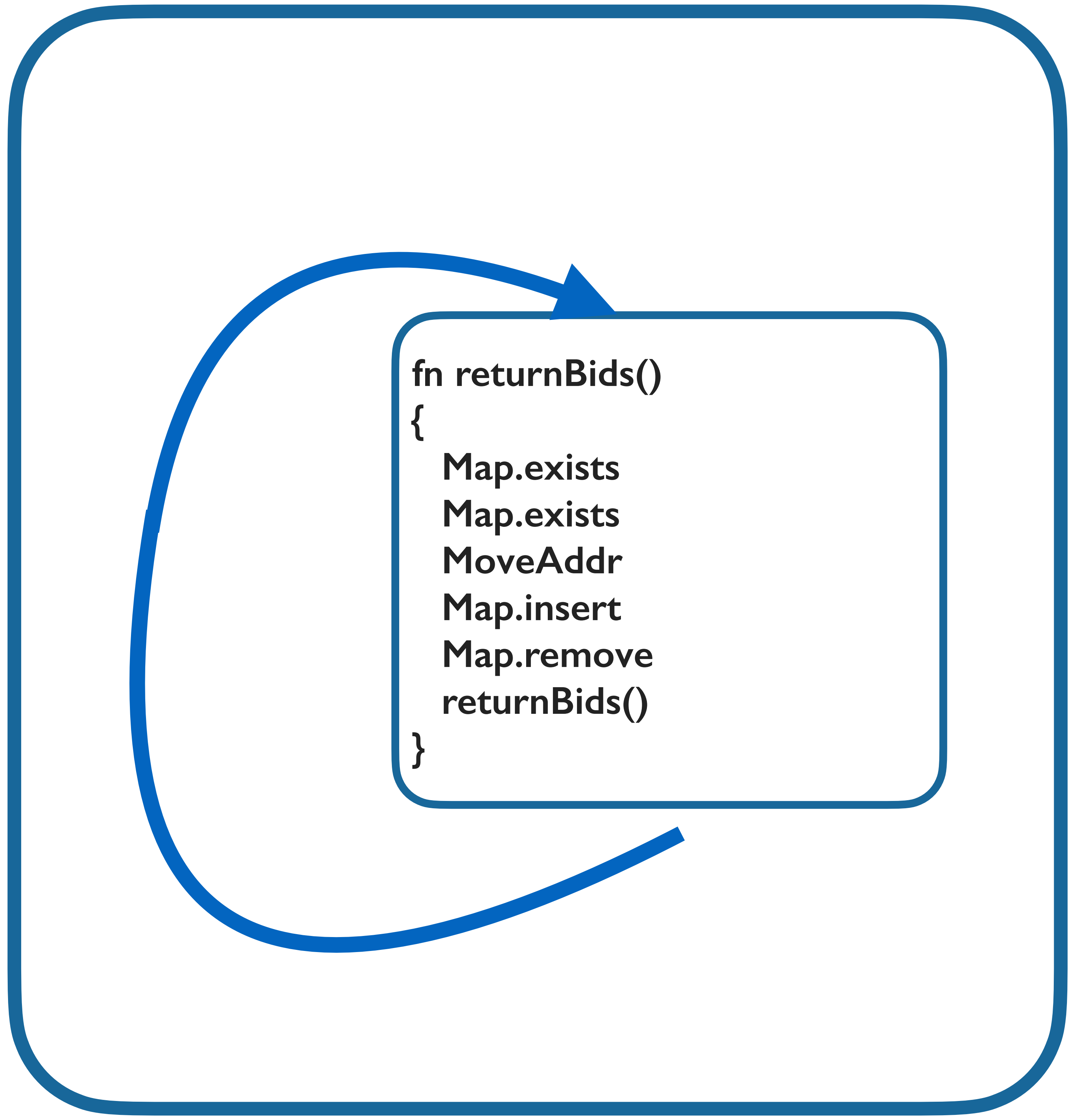


Less costly branch augmented with Gas.deposit

Hoare-logic style gas-analysis framework

inference of exact gas bounds in linear-time

low programmer burden
no gas tracking



Gas amortization pays for the cost of recursion