A Language-based Serverless Function Accelerator

Emily Herbert

What is serverless computing?

Approach to cloud computing…

without servers…

with servers

Google Cloud Functions

APACHE OpenWhisk™

1. Writes the application code

2. Manages the cloud infrastructure
   a. operating system
   b. firewall
   c. load balancer
   d. web server
   e. file server

- security
- fault tolerance
- resource allocation

traditional cloud computing setting

1. Writes a “serverless function”

- cloud infrastructure is completely hidden

- security
- fault tolerance
- resource allocation

1. Manages the cloud infrastructure
   a. operating system
   b. firewall
   c. load balancer
   d. web server
   e. file server

- security
- fault tolerance
- resource allocation

serverless computing setting

serverless function

+

- cloud infrastructure is completely hidden

cloud provider's infrastructure is elastic and volatile

Google Cloud Functions

invoker ← web requests

- function instances start and stop to meet demands

serverless computing setting

```
3  const request = require('request');
4
5  const url = "https://api.census.gov/data/timeseries/asm/industry?get=NAICS_TTL,EM
6
7  function censusdata(callback) {
8    request({
9      url: url,
10     json: true
11   }, function (error, response, body) {
12     if (!error && response.statusCode === 200) {
13       const tupleData = body.slice(1).map(function(row) {
14         return {
15           "Jobs": row[1],
16           "Year": row[3]
17         };
18       });
19
20       return callback(JSON.stringify(tupleData), response.statusCode);
21     } else {
22       return callback(error, response.statusCode);
23     }
24   });
25 }
26
27 exports.main = function(req, res) {
28   censusdata(function(output, status) {
29     res.set("content-type", status === 200 ? "application/json" : "text/plain");
30     res.status(status).send(output);
31   });
32 };
```

sending out a request

returning a response containing census data

entry point

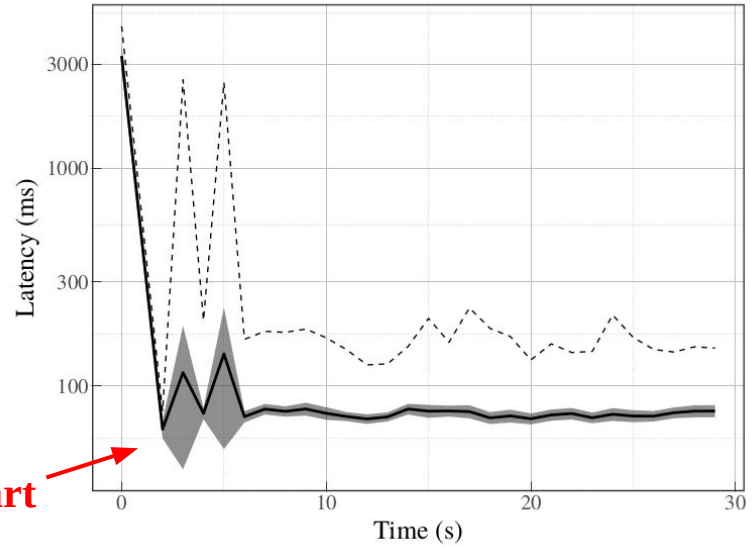
cloud provider's infrastructure is elastic and volatile

1. Idempotent (tolerant to re-execution)
2. Transient in-memory state
3. Short-lived
4. Consume limited memory

Performance experiment

```
exports.hello = (req, res) => {
   res.send('Hello World!');
 };
```

- Hosted on Google Cloud Functions
- Requests sent from 10 open connections for 30 seconds



cold start

docker

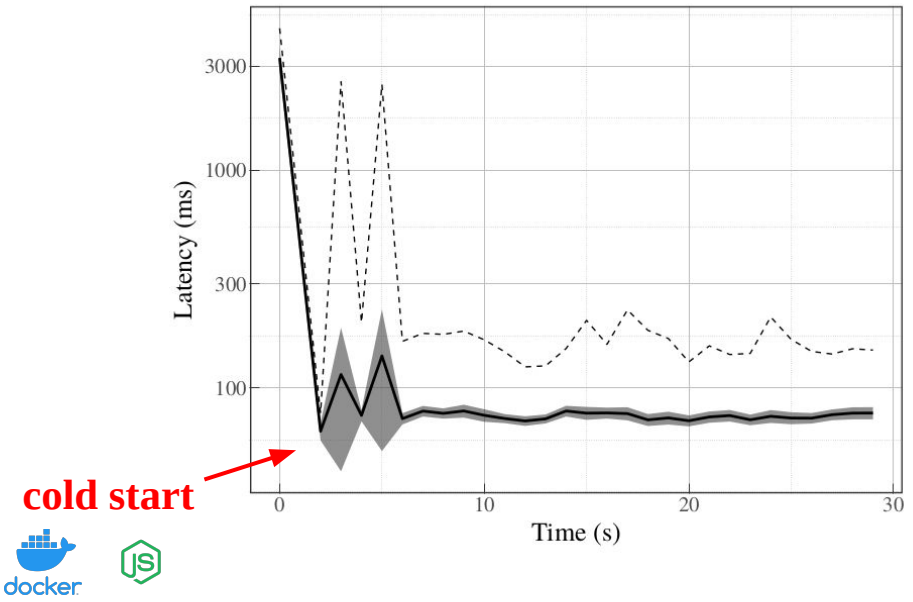
Performance experiment

```
exports.hello = (req, res) => {
   res.send('Hello World!');
 };
```

- Hosted on Google Cloud Functions
- Requests sent from 10 open connections for 30 seconds

1. Significant cold starts
   (> 10x exec time for short functions) [1]

2. Slowdown from containerization
   (up to 20x slowdown from native exec) [1]



[1] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In IEEE/ACM International Symposium on Microarchitecture (MICRO).

Rust as an alternative

- Type system with memory safety guarantees
  - no dangling pointers
  - no use-after-frees
  - no undefined behavior

- A serverless platform that runs Rust functions? [2]

- Can run multiple functions in one process using **language-based isolation**

Microservices Resident?	Isolation	Latency (µs) Median	99%	Throughput (M invoc/s)
Warm-start	Process	8.7	27.3	0.29
	Language	1.2	2.0	5.4
Cold-start	Process	2845.8	15976.0	–
	Language	38.7	42.2	–

Table 1: Microservice invocation performance

[2] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. 2018. Putting the "Micro" back in microservices. In USENIX Annual Technical Conference (ATC).

Rust as an alternative

- Difficult to learn for the average web programmer
- Programmers might not be looking to learn a new language
- Does not prevent:
  - CPU monopolization
  - deadlocks
  - memory leaks
  - …

How do we remedy this?
… Containerless!

1. Containerless overview
2. Building traces
3. Functions
4. Evaluation

Containerless

- “Serverless function accelerator” that seeks to improve performance
- Uses **language-based isolation** when possible, and container-based isolation if necessary
- Prevents CPU monopolization and places memory limits

AWS Lambda

Google Cloud Functions

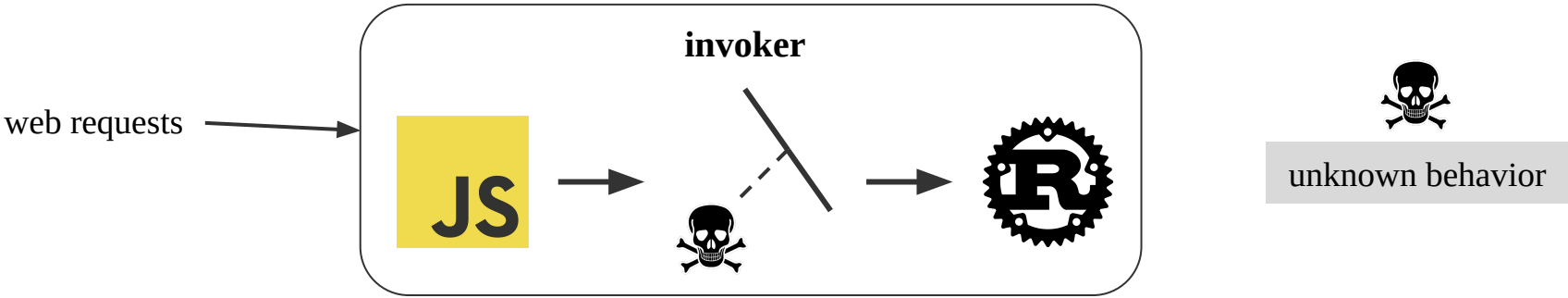
APACHE OpenWhisk™

★ can use language of choice
★ benefit from lower response latency

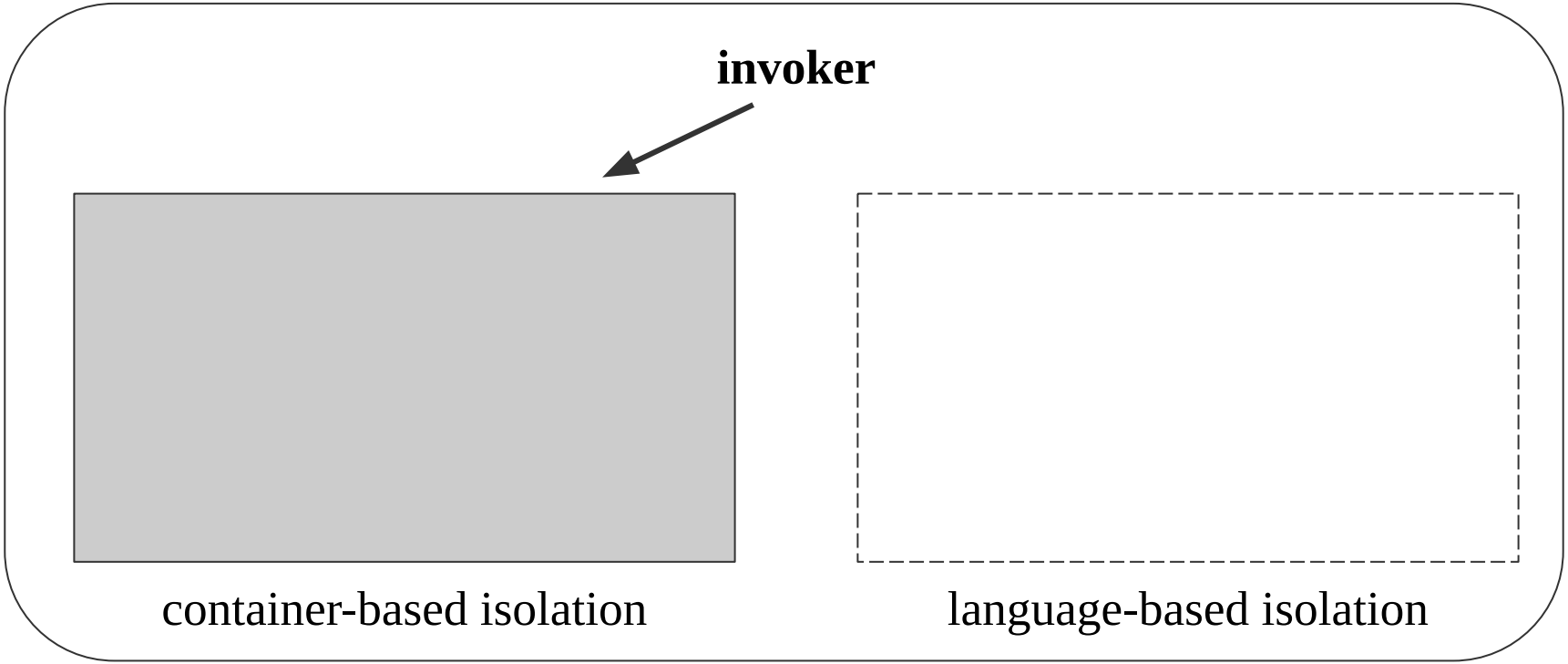
★ benefit from lower resource utilization
★ can share idle resources across all customers

Containerless

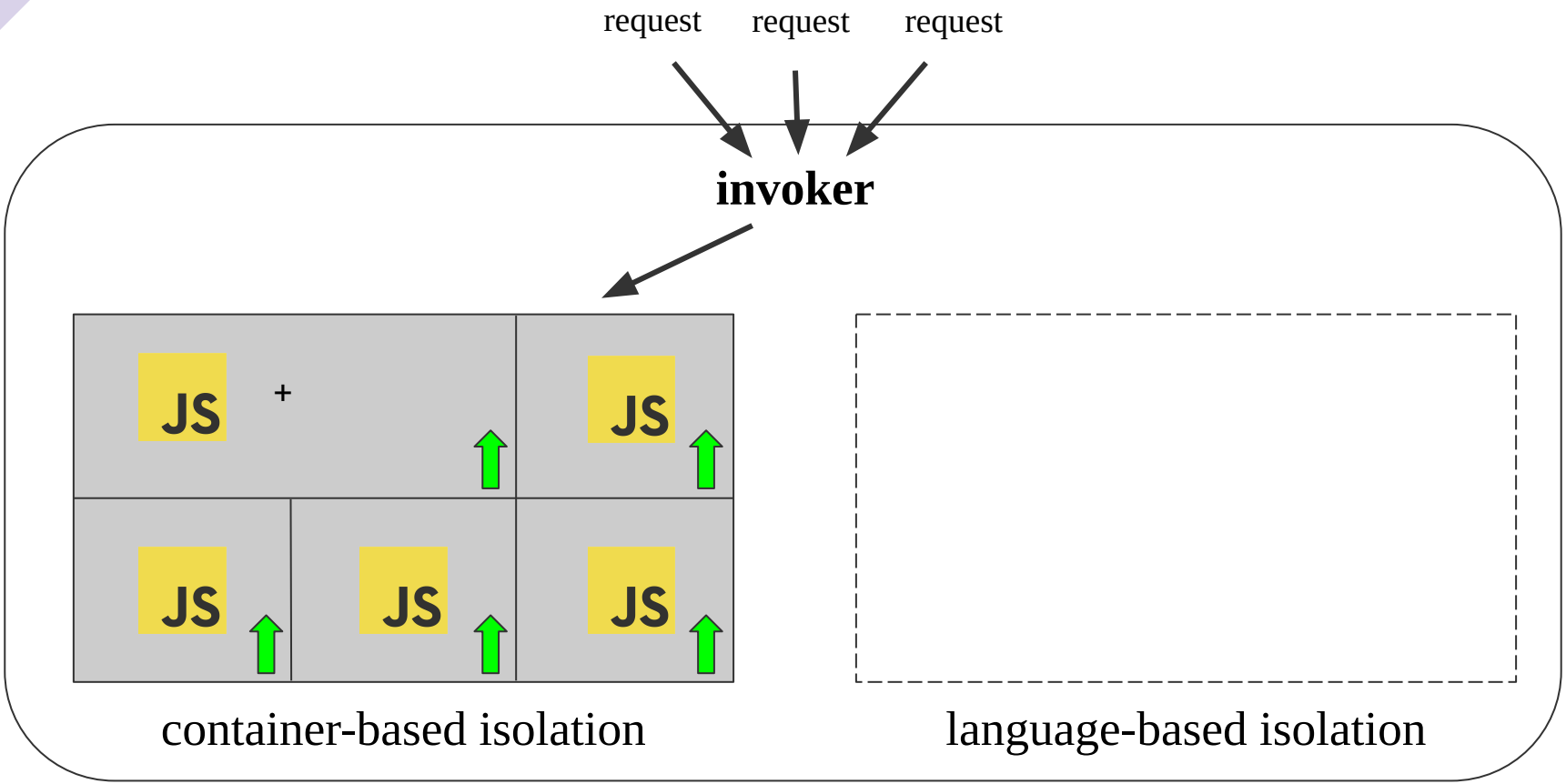
- Transforms JavaScript to Rust by means of a **trace-based representation**
- Traces are built incrementally at runtime, and feature the possibility of **unknown behavior**
- Employs the Rust type system to ensure memory-safety (**language-based isolation**)
- Uses container-based isolation as a safefall



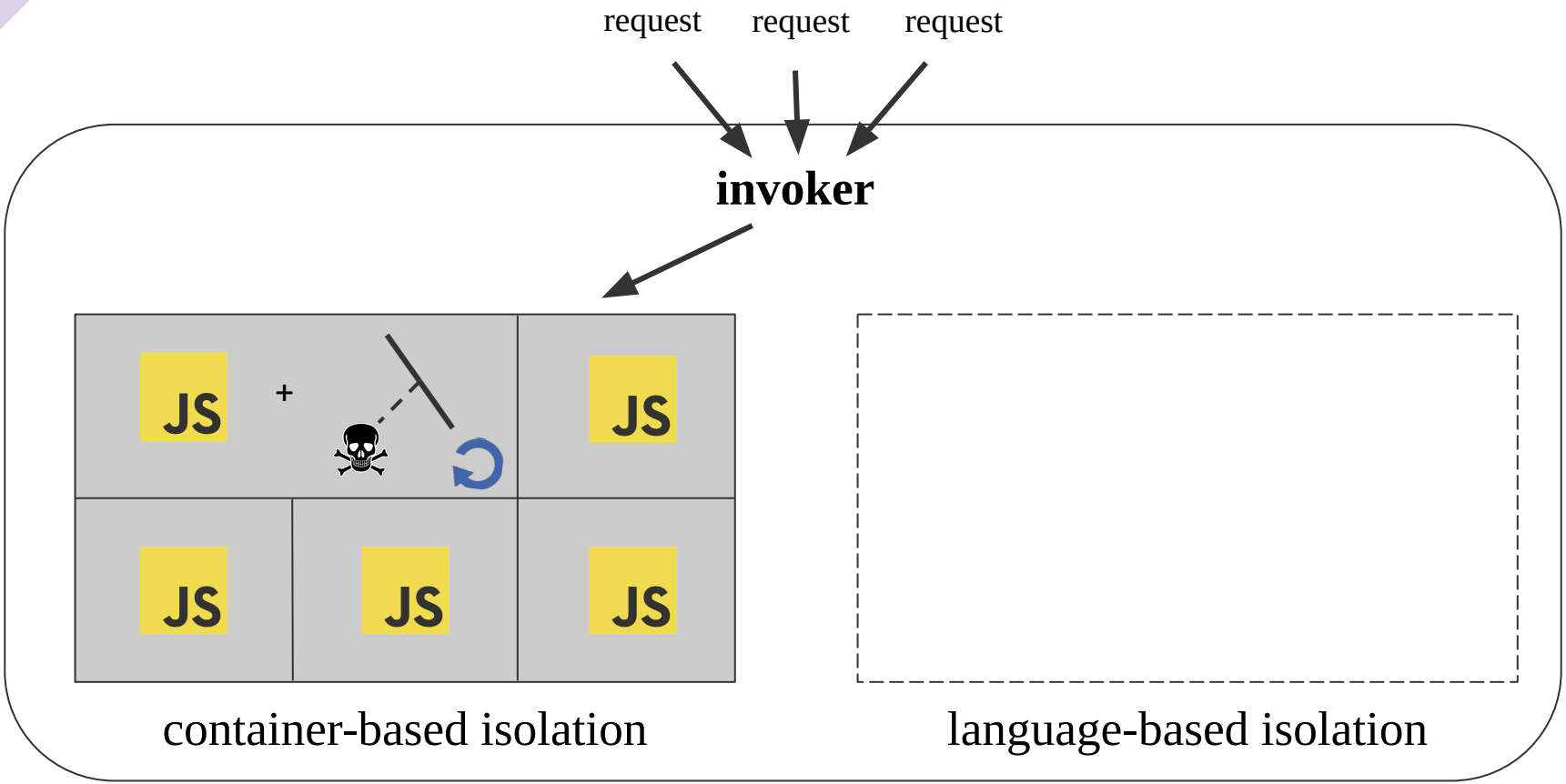
Containerless



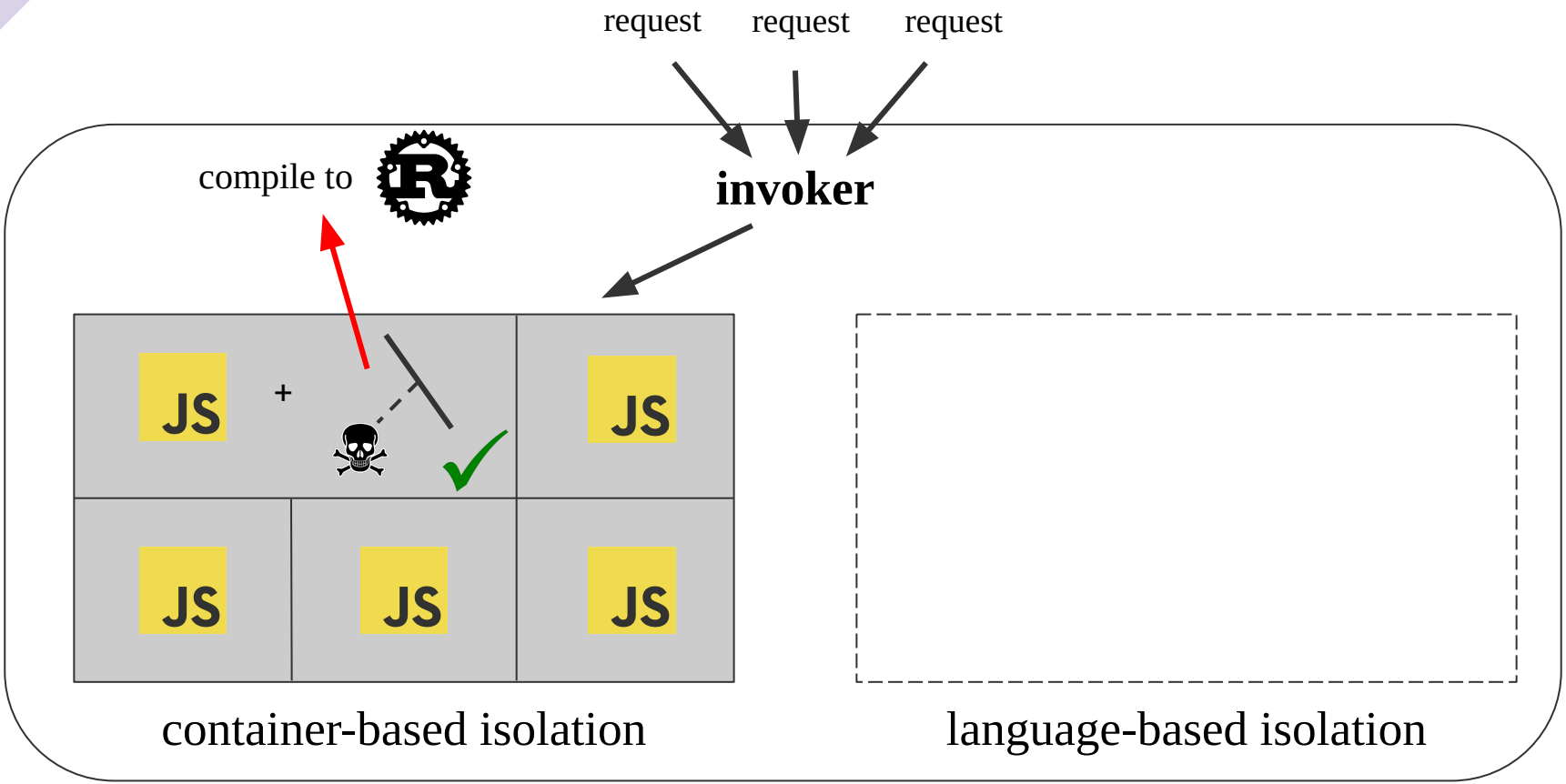
Containerless



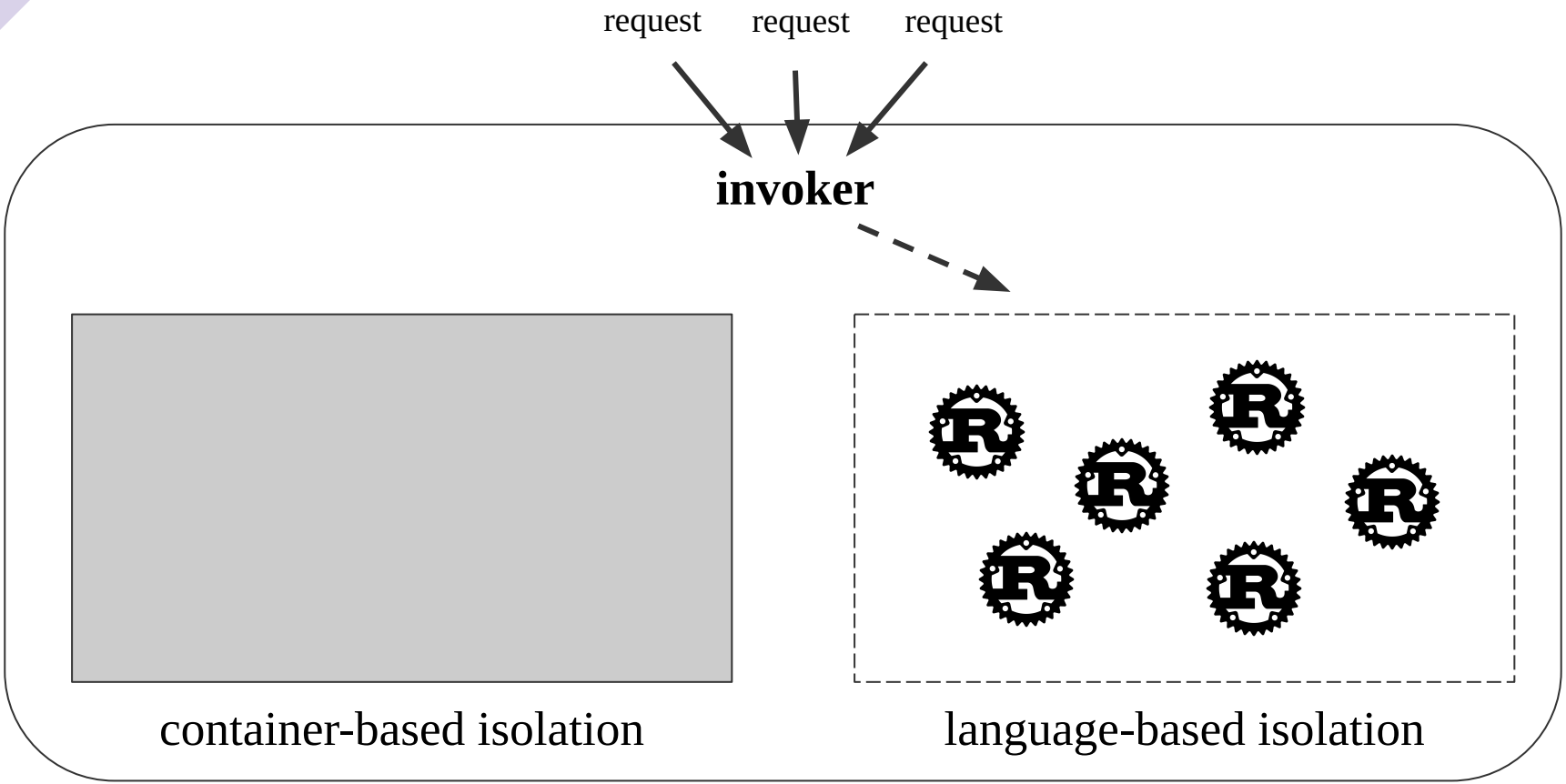
Containerless



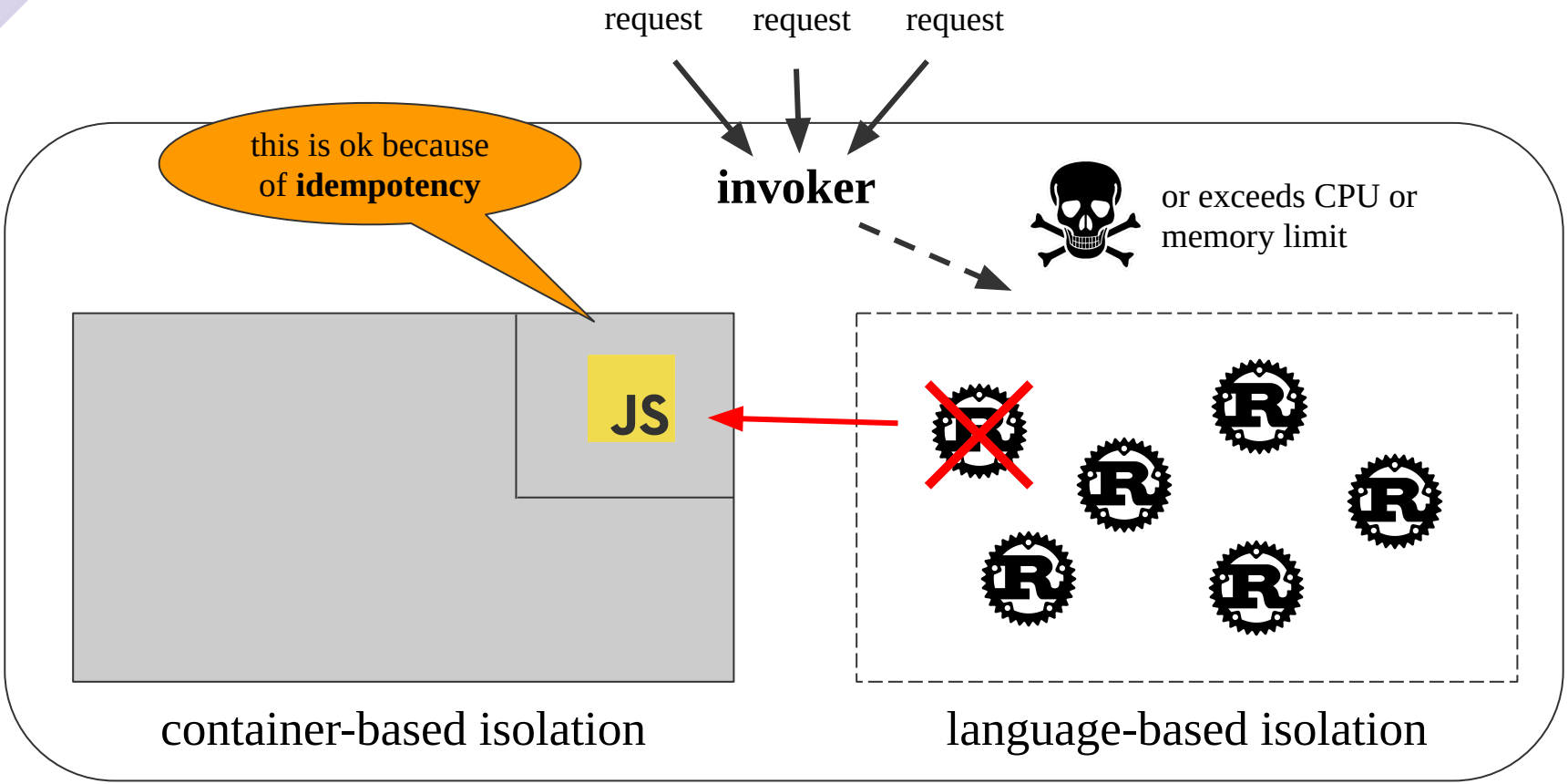
Containerless



Containerless

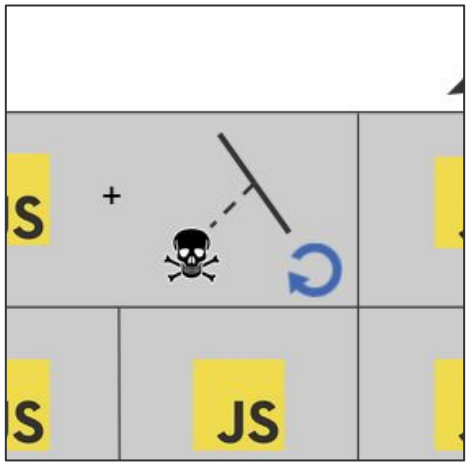


Containerless



Containerless

1. Containerless overview
2. Building traces
3. Functions
4. Evaluation



Building Traces

- We want to build execution traces incrementally at runtime
- We want the ability to express **unknown execution paths**
- Thus, we create a **trace language** and build traces at runtime
- Simplified subset:

l-values

$tlv ::= x$	Variable

Blocks

$tblk ::= \{\ t_1 \cdots t_n \}$

Trace trees

$t ::= c$	Constant
$\mid x$	Variable
$\mid t_1\ op\ t_2$	Binary operation
$\mid tblk$	Block
$\mid$ **if** $(t_1)\ t_2$ **else** t_3	Conditionals
$\mid$ **while** $(t_1)\ tblk$	Loops
$\mid$ **let** $x = t$;	Variable declaration
$\mid tlv = t$;	Assignment
$\mid tblk$	Block
$\mid$ ☠	Unknown behavior

Trace State

- We want to build execution traces incrementally at runtime
- We need a mechanism of tracing the currently executing statement
- Thus, we introduce the **trace state** (c)
- When tracing begins, we initialize the trace state to the unknown statement 
- Example trace states:

```
let x = 10;
```

```
if (x<0) {
  y = x * -1;
} else {
}
```

```
if (x<0) {
  y = x * -1;
} else {
  y = x;
}
```

Trace Context

- We need a mechanism of identifying our current position in the trace
- We need the ability to **merge traces** from multiple executions
- Thus, we introduce the **trace context** (κ)

trace context ≠ evaluation context

- Simplified subset:

conditional false branch previous context

$\kappa ::= \cdot$	Empty context
$\mid \textsc{IfTrue}(t_1, t_2, \kappa)$	In the true branch of an *if*, with condition t_1 and false branch t_2.
$\mid \textsc{IfFalse}(t_1, t_2, \kappa)$	In the false branch of an *if*, with condition t_1 and true branch t_2.
$\mid \textsc{While}(t, \kappa)$	In the body of a loop, with condition t.

- a trace context (κ) is a representation of a trace with a “hole”

while $(y<0)$ {**if** $(x>0)$ □ **else** ☠}

$\textsc{IfTrue}(x>0, ☠, \textsc{While}(y<0, \cdot))$

if we are currently executing+tracing this part

then this is the current trace context

x = 5
x = -5

```
1 if (x<0) {
2   y = x * -1;
3 } else {
4   y = x;
5 }
```

trace context (κ)	trace state (c)
· (“empty”)	☠ (“unknown”)
IfTrue(x<0, ☠, ·)	☠
IfTrue(x<0, ☠, ·)	y = x * -1;

·	if (x<0) {
	  y = x * -1;
	} else {
	  ☠
	}

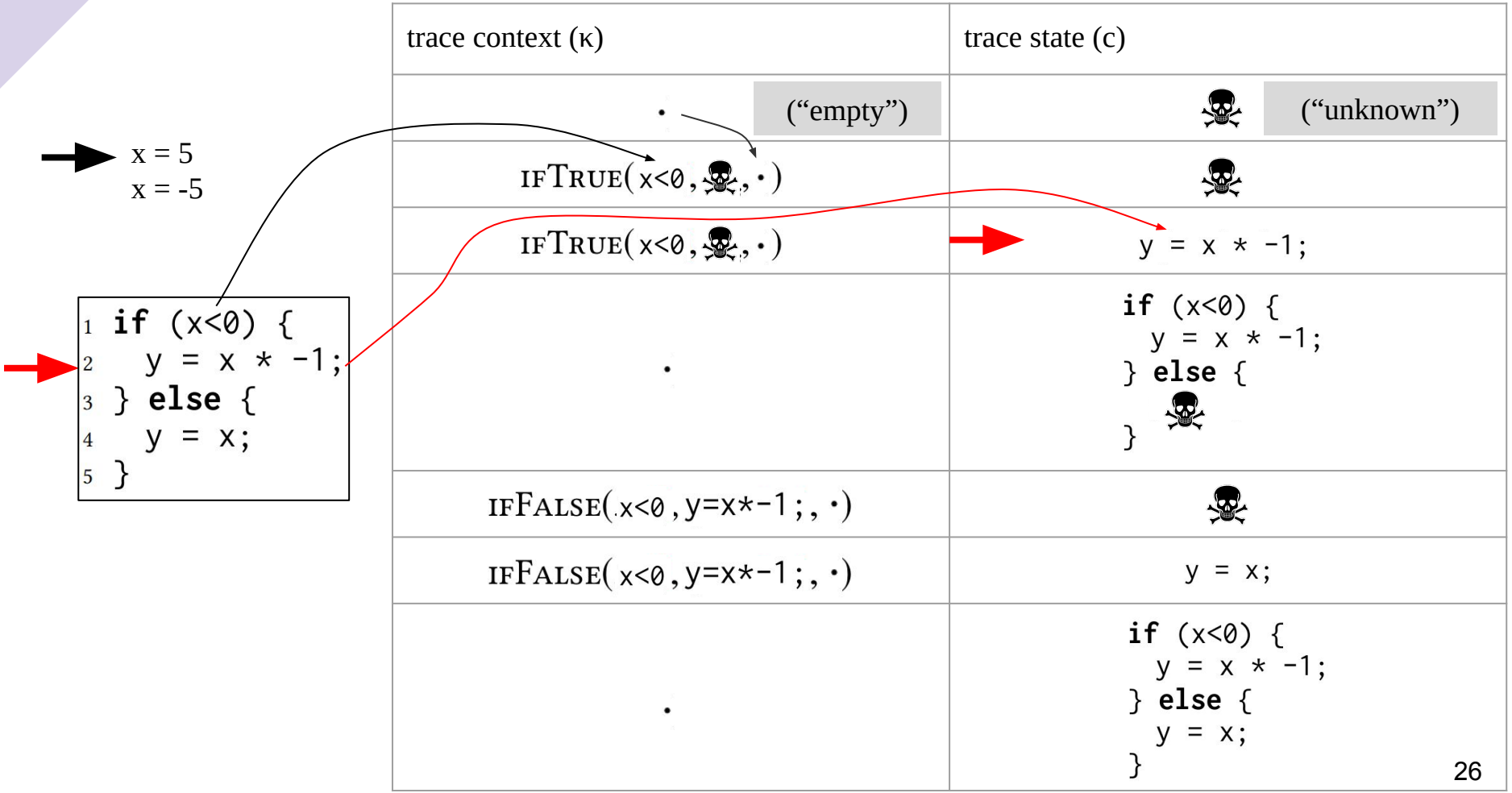
IfFalse(x<0, y=x*-1;, ·)	☠
IfFalse(x<0, y=x*-1;, ·)	y = x;

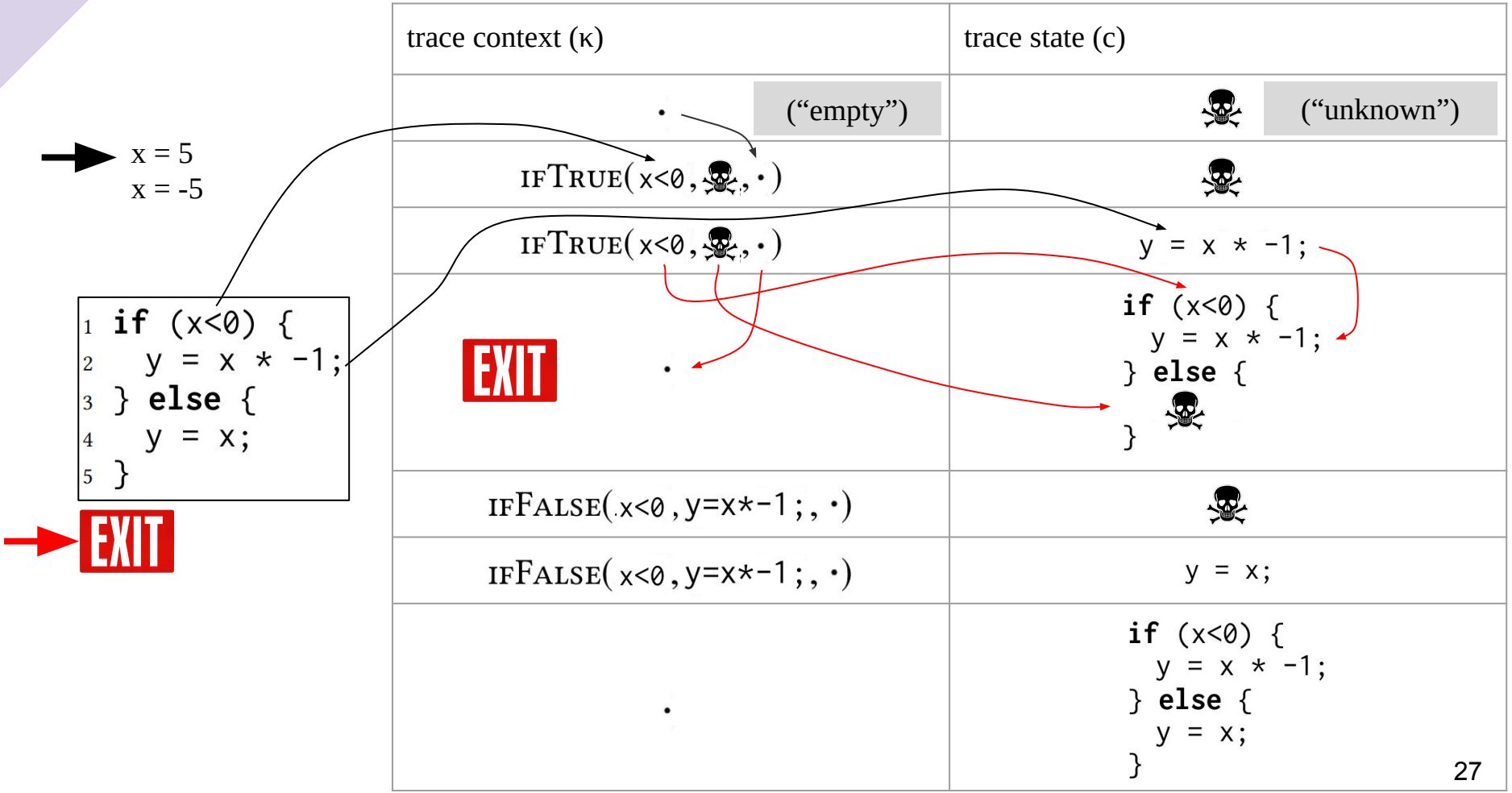
·	if (x<0) {
	  y = x * -1;
	} else {
	  y = x;
	}

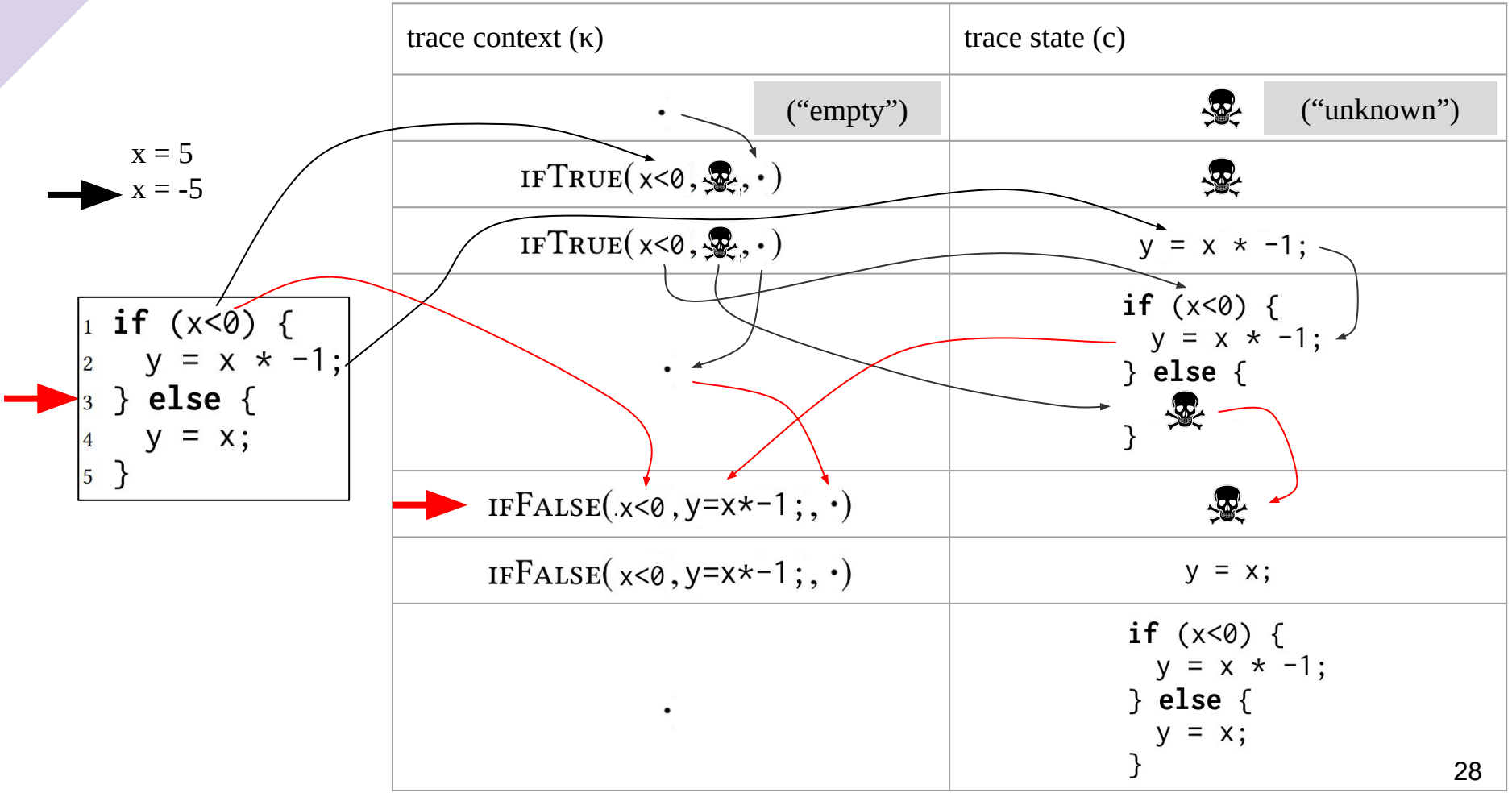
x = 5
x = -5

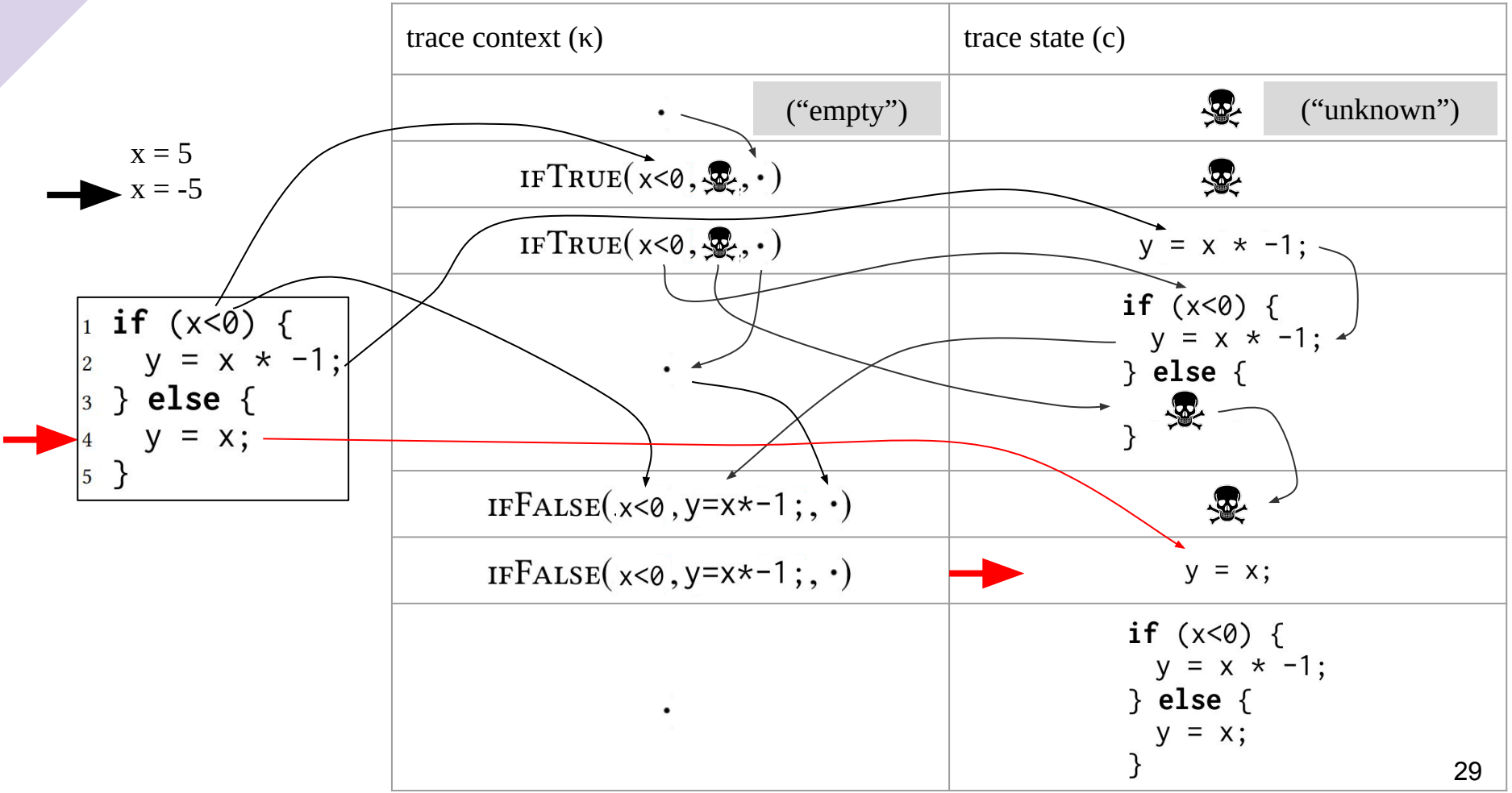
```
1 if (x<0) {
2   y = x * -1;
3 } else {
4   y = x;
5 }
```

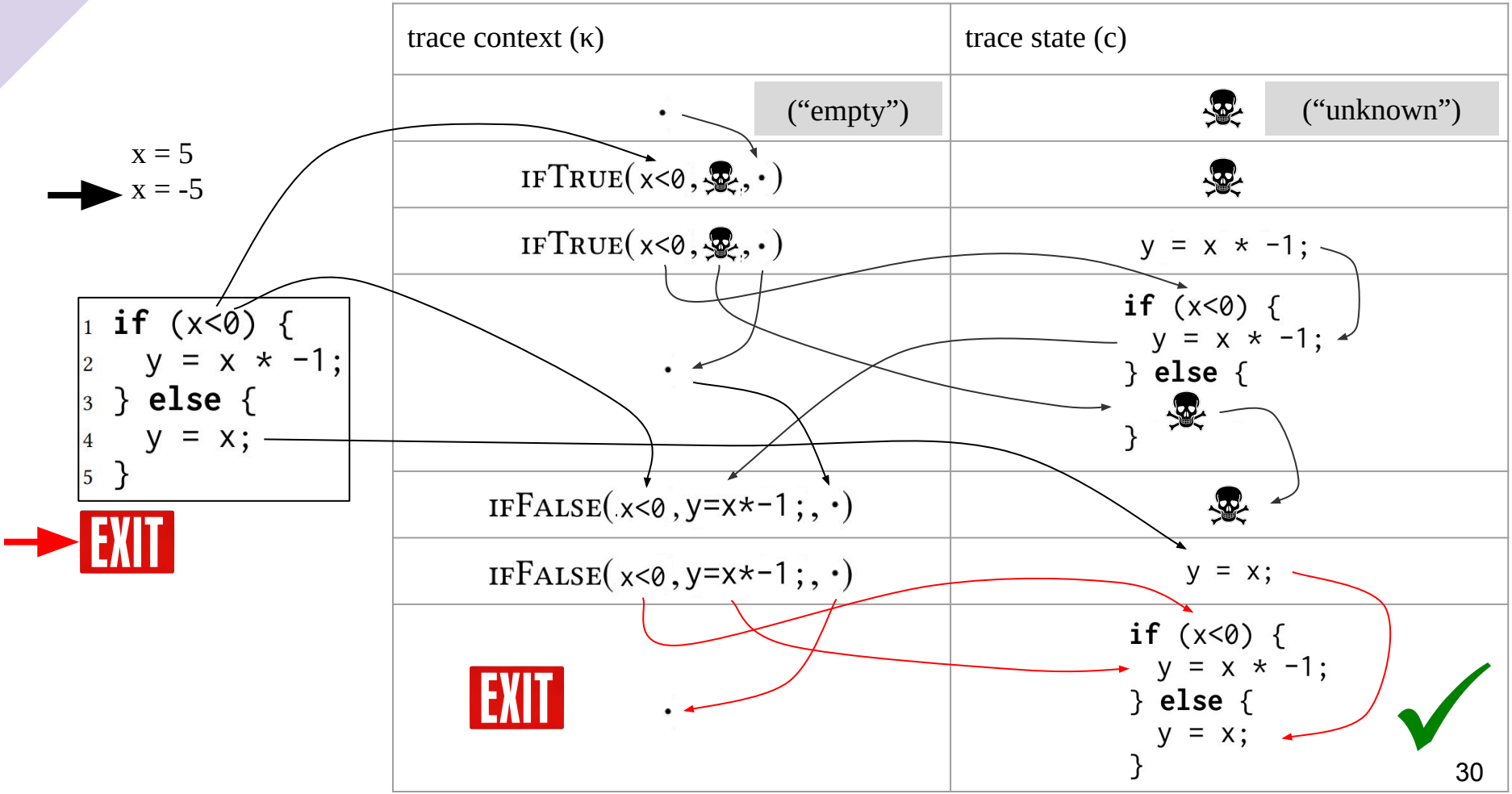
trace context (κ)	trace state (c)
false branch · (“empty”)	☠ (“unknown”)
IFTRUE(x<0, ☠, ·)	☠
IFTRUE(x<0, ☠, ·)	`y = x * -1;`
·	`if (x<0) { y = x * -1; } else { ☠ }`
IFFALSE(x<0, y=x*-1;, ·)	☠
IFFALSE(x<0, y=x*-1;, ·)	`y = x;`
·	`if (x<0) { y = x * -1; } else { y = x; }`


x = 5
x = -5
1 if (x<0) {
2 y = x * -1;
3 } else {
4 y = x;
5 }
trace context (κ)
trace state (c)
(“empty”)
(“unknown”)
IFTRUE(x<0, , ·)
IFTRUE(x<0, , ·)
y = x * -1;
if (x<0) {
y = x * -1;
} else {
}
IFFALSE(x<0, y=x*-1;, ·)
IFFALSE(x<0, y=x*-1;, ·)
y = x;
if (x<0) {
y = x * -1;
} else {
y = x;
}


x = 5
x = -5
1 if (x<0) {
2 y = x * -1;
3 } else {
4 y = x;
5 }
EXIT
trace context (κ)
trace state (c)
(“empty”)
(“unknown”)
IFTRUE(x<0, , ·)
IFTRUE(x<0, , ·)
y = x * -1;
if (x<0) {
y = x * -1;
} else {
}
EXIT
IFFALSE(x<0 , y=x*-1; , ·)
IFFALSE(x<0 , y=x*-1; , ·)
y = x;
if (x<0) {
y = x * -1;
} else {
y = x;
}

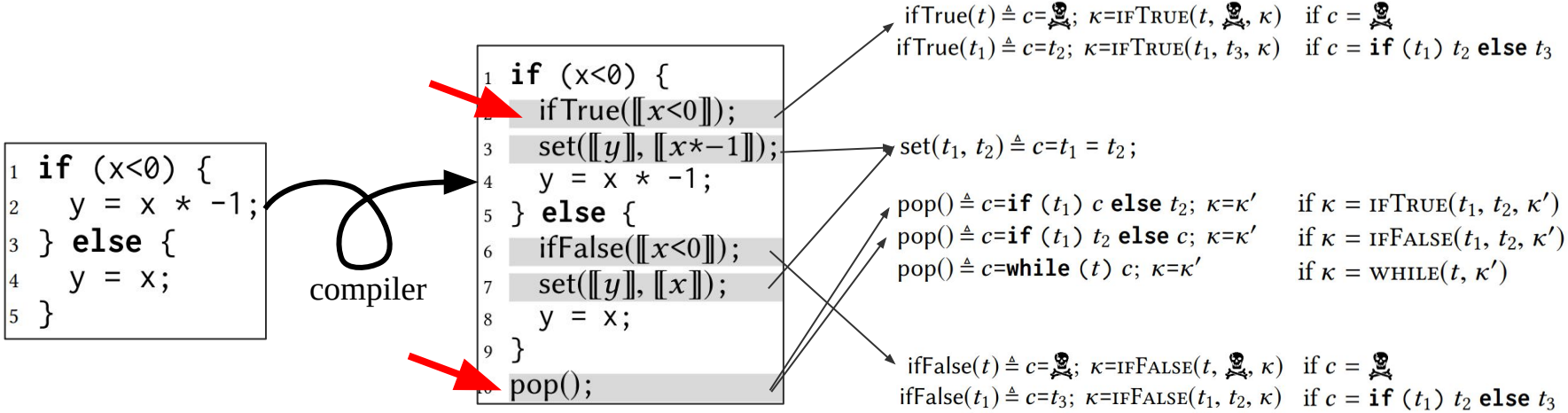

trace context (κ)
trace state (c)
(“empty”)
(“unknown”)
x = 5
x = -5
1 if (x<0) {
2 y = x * -1;
3 } else {
4 y = x;
5 }
IFTRUE(x<0 , , ·)
IFTRUE(x<0 , , ·)
y = x * -1;
if (x<0) {
y = x * -1;
} else {
}
IFFALSE(x<0 , y=x*-1; , ·)
IFFALSE(x<0 , y=x*-1; , ·)
y = x;
if (x<0) {
y = x * -1;
} else {
y = x;
}


trace context (κ)
trace state (c)
(“empty”)
(“unknown”)
x = 5
x = -5
1 if (x<0) {
2 y = x * -1;
3 } else {
4 y = x;
5 }
IFTRUE(x<0, ☠, ·)
IFTRUE(x<0, ☠, ·)
y = x * -1;
if (x<0) {
y = x * -1;
} else {
}
IFFALSE(x<0, y=x*-1;, ·)
IFFALSE(x<0, y=x*-1;, ·)
y = x;
if (x<0) {
y = x * -1;
} else {
y = x;
}


trace context (κ)
trace state (c)
(“empty”)
(“unknown”)
x = 5
x = -5
1 if (x<0) {
2 y = x * -1;
3 } else {
4 y = x;
5 }
EXIT
IFTRUE(x<0, , ·)
IFTRUE(x<0, , ·)
y = x * -1;
if (x<0) {
y = x * -1;
} else {
}
IFFALSE(x<0, y=x*-1;, ·)
IFFALSE(x<0, y=x*-1;, ·)
y = x;
EXIT
if (x<0) {
y = x * -1;
} else {
y = x;
}

Trace Compiler + Runtime System

- Trace compiler instruments calls to a trace-building runtime system
- The example simplifies things a bit
- Alternative approach: modify node interpreter to build traces

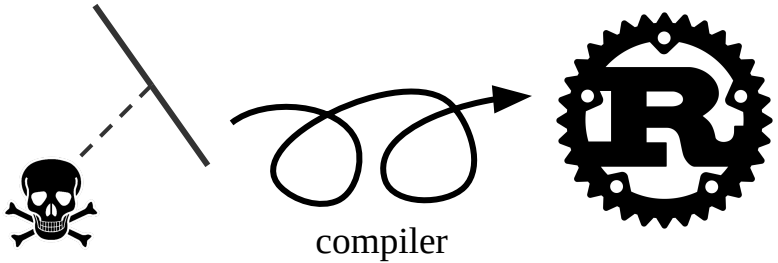


runtime system

Trace-to-Rust Compiler

this is ok because serverless functions tolerate **transient memory**

- Resolves mismatch issues between traces and Rust
- JavaScript has garbage collection and Rust does not, so we **add arena allocation**
- Traces allow variable aliasing and Rust does not, so we **wrap variables in a container type with dynamic ownership rules**
- Traces are dynamically typed and Rust is statically typed, so we **inject all values into a dynamic type**



1. Containerless overview
2. Building traces
3. Functions
4. Evaluation

Tracing Functions

- User-defined functions are difficult to translate to Rust directly, because of Rust's lifetime and ownership rules
- We **eliminate functions** by introducing an expression that represents a function environment and by introducing addresses
- We **inline function application** using labels and breaks

l-values

$tlv ::= x$	Variable
$\mid *t\,.\,x$	Variable in environment

Addresses

$a ::= t\,.\,x$	Address in environment
$\mid \&x$	Address of variable

Blocks

$tblk ::= \{\ t_1 \cdots t_n\}$

Trace trees

$t ::= c$	Constant
$\mid x$	Variable
$\mid t_1\ op\ t_2$	Binary operation
$\mid tblk$	Block
$\mid$ **if** $(t_1)\ t_2$ **else** t_3	Conditionals
$\mid$ **while** $(t_1)\ tblk$	Loops
$\mid$ **let** $x = t\,;$	Variable declaration
$\mid tlv = t\,;$	Assignment
$\mid tblk$	Block
$\mid \ell : t$	Labelled trace
$\mid$ ☠	Unknown behavior
$\mid$ **break** $\ell\ t\,;$	Break with value
$\mid$ **env**$(x_1 : a_1, \cdots, x_n : a_n)$	Environment object
$\mid *t\,.\,x$	Value in environment

Tracing Functions

- The trace context is expanded to include named values (function application):

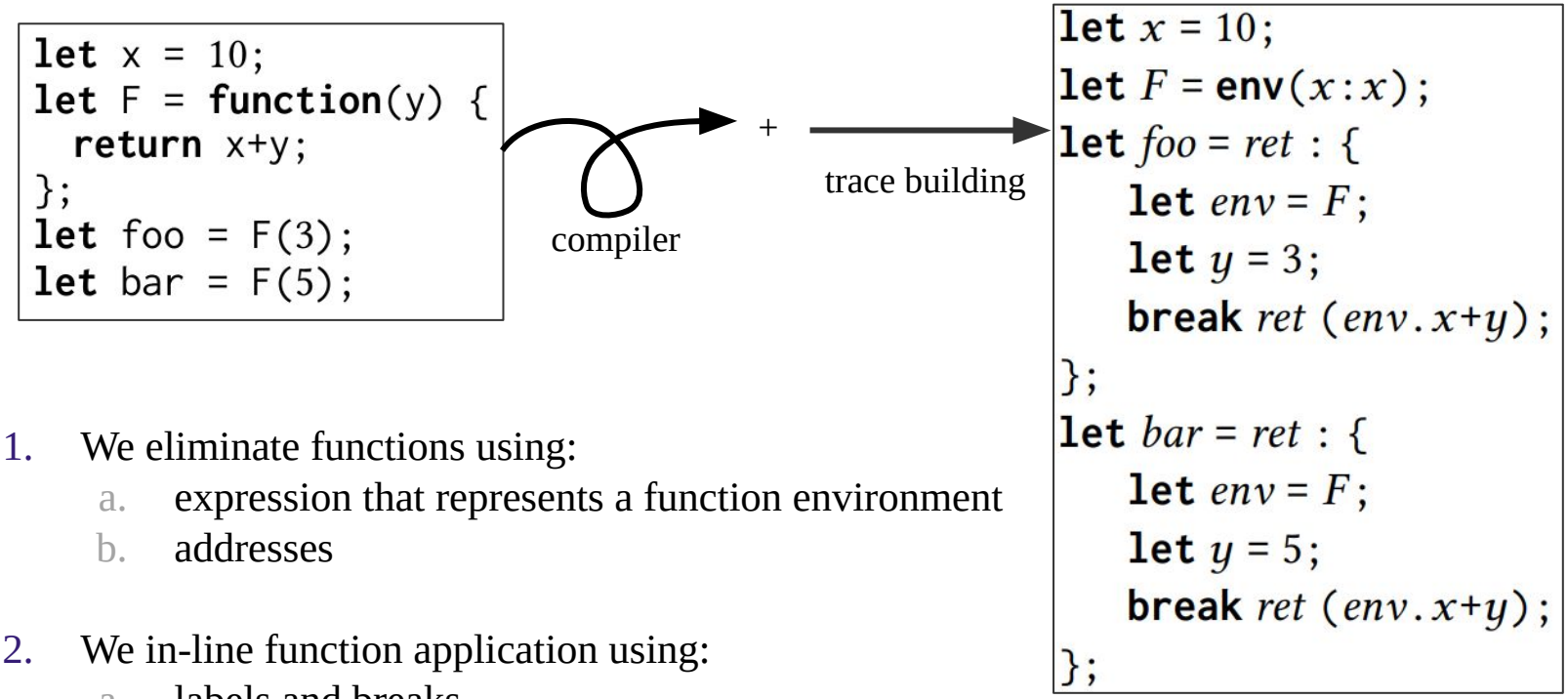
$\kappa ::= \cdot$	Empty context
$\mid$ SEQ($[t_1 \cdots t_{i-1}]$, $[t_{i+1} \cdots t_n]$, κ)	In a block, with $[t_1 \cdots t_{i-1}]$ already executed.
$\mid$ IFTRUE(t_1, t_2, κ)	In the true branch of an *if*, with condition t_1 and false branch t_2.
$\mid$ IFFALSE(t_1, t_2, κ)	In the false branch of an *if*, with condition t_1 and true branch t_2.
$\mid$ WHILE(t, κ)	In the body of a loop, with condition t.
$\mid$ LABEL(ℓ, κ)	In the body of a labeled trace, with label ℓ.
$\mid$ NAMED(x, κ)	In the body of a named variable x.

variable name **previous context**

```
1 if (x<0) {
2   y = x * -1;
3 } else {
4   y = x;
5 }
```

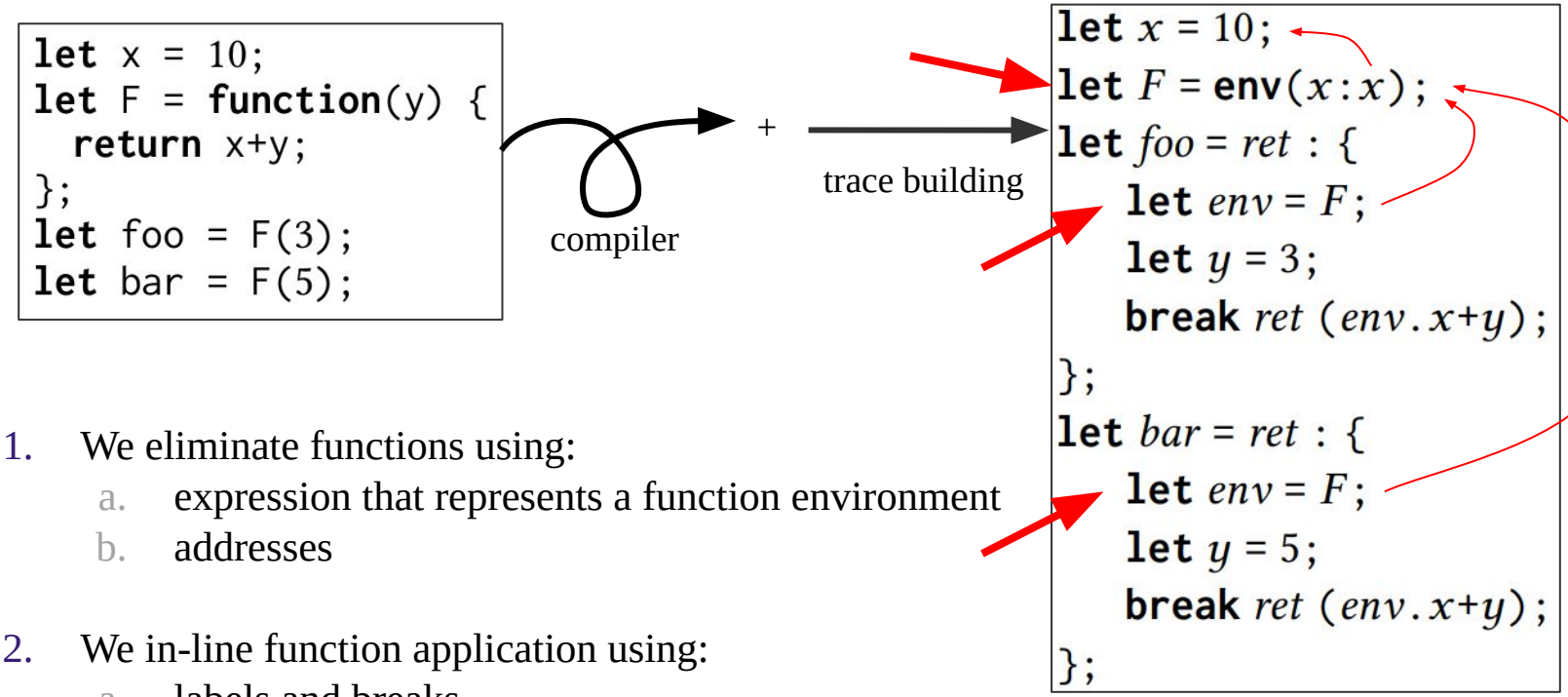
desugaring to ANF
+
compiler

```
1 if (x<0) {
2   ifTrue(⟦x<0⟧);
3   set(⟦y⟧, ⟦x*-1⟧);
4   y = x * -1;
5 } else {
6   ifFalse(⟦x<0⟧);
7   set(⟦y⟧, ⟦x⟧);
8   y = x;
9 }
10 pop();
```



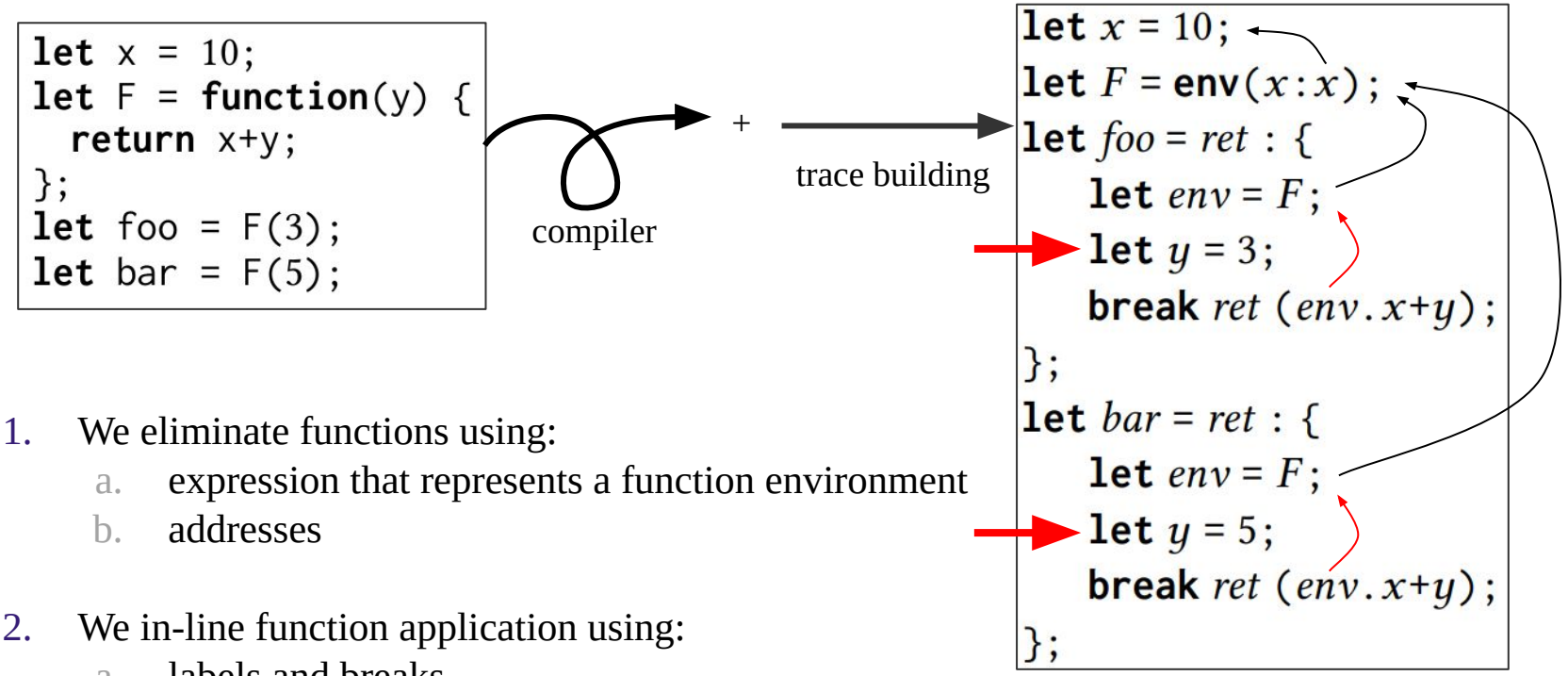
1. We eliminate functions using:
   a. expression that represents a function environment
   b. addresses

2. We in-line function application using:
   a. labels and breaks
   b. **shadow argument stack** that tracks the traced representations of function arguments



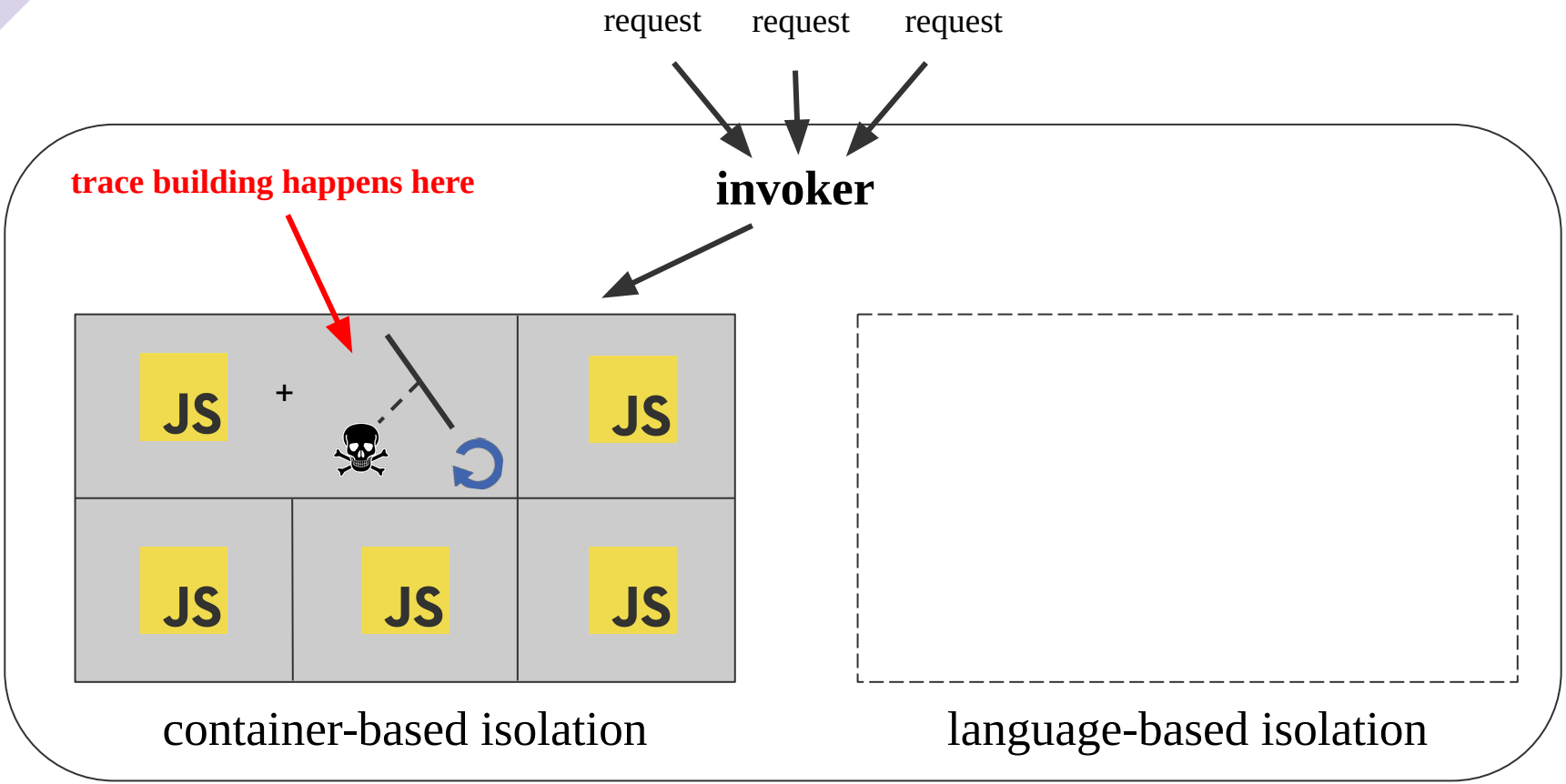
1. We eliminate functions using:
   a. expression that represents a function environment
   b. addresses

2. We in-line function application using:
   a. labels and breaks
   b. **shadow argument stack** that tracks the traced representations of function arguments



1. We eliminate functions using:
   a. expression that represents a function environment
   b. addresses

2. We in-line function application using:
   a. labels and breaks
   b. **shadow argument stack** that tracks the traced representations of function arguments



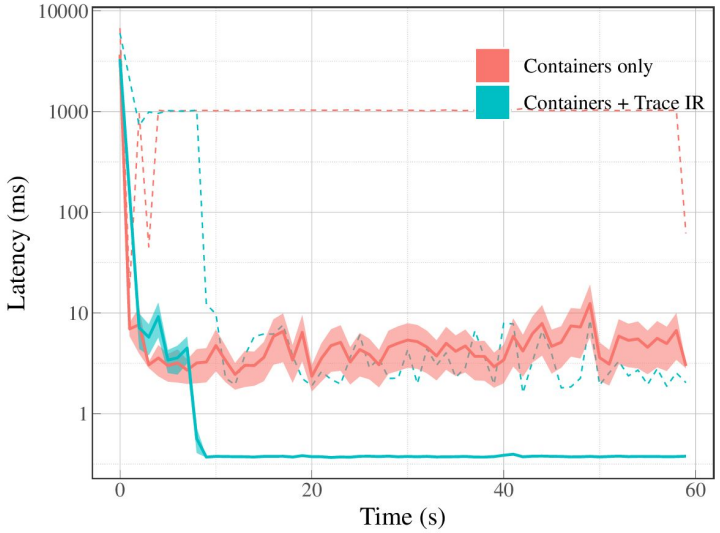
Containerless

1. Containerless overview
2. Building traces
3. Functions
4. Evaluation

Evaluation

- 6 benchmarks
- Each evaluated with requests from 10 concurrent open connections for 60 seconds

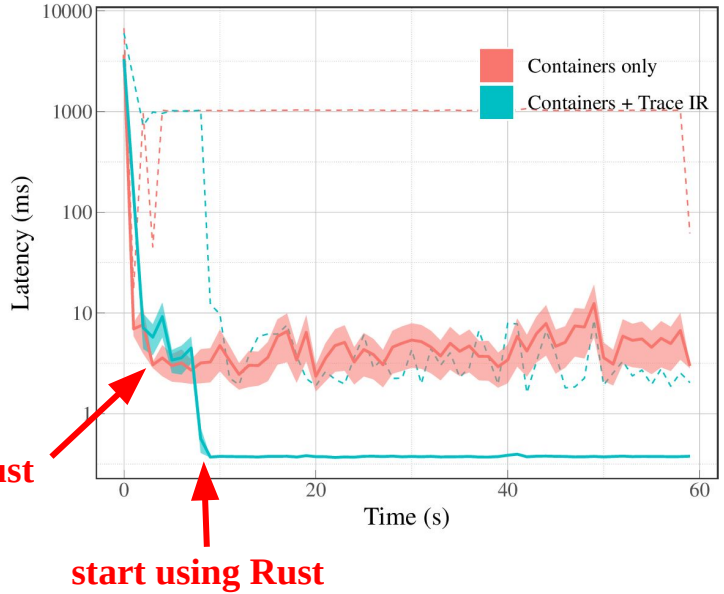
authorize

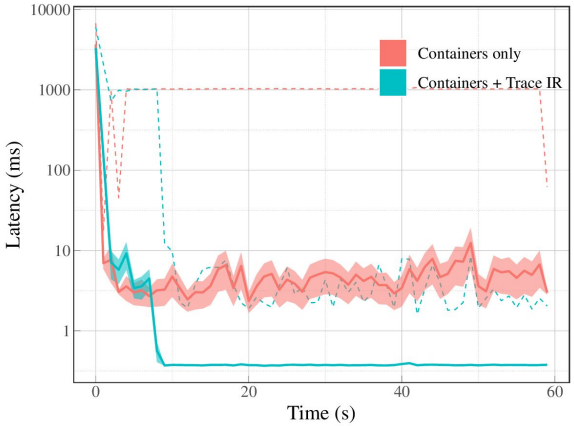


Evaluation

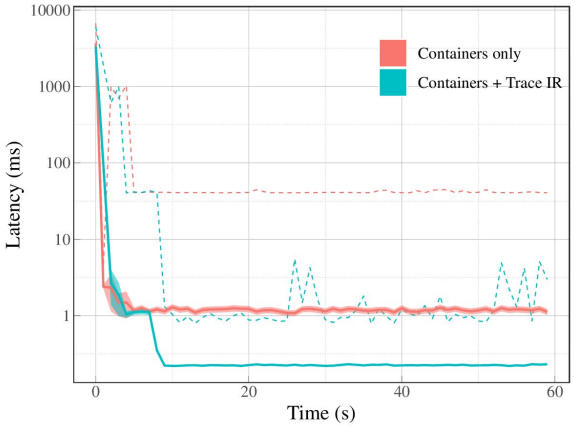
- 6 benchmarks
- Each evaluated with requests from 10 concurrent open connections for 60 seconds

authorize

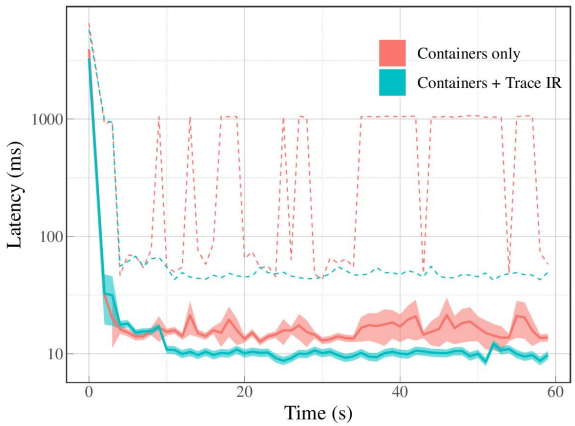




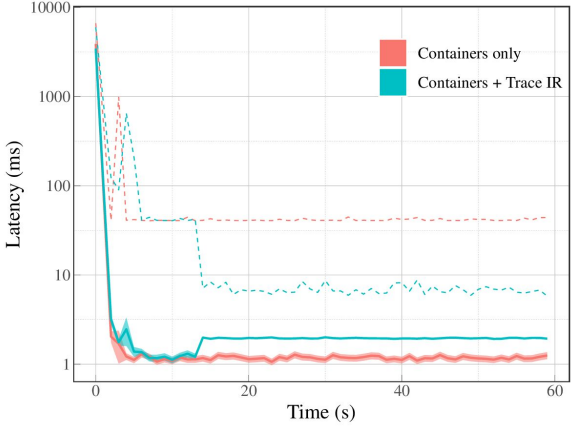
authorize



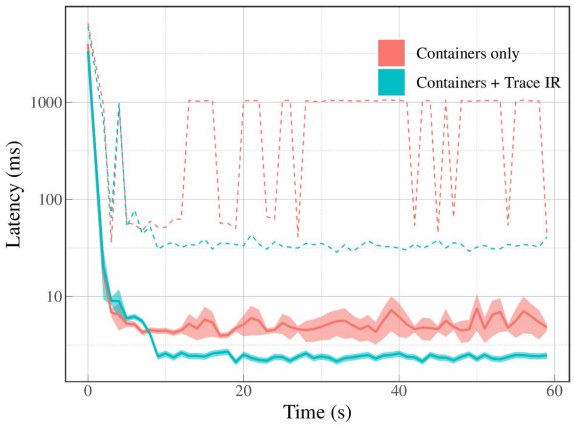
autocomplete



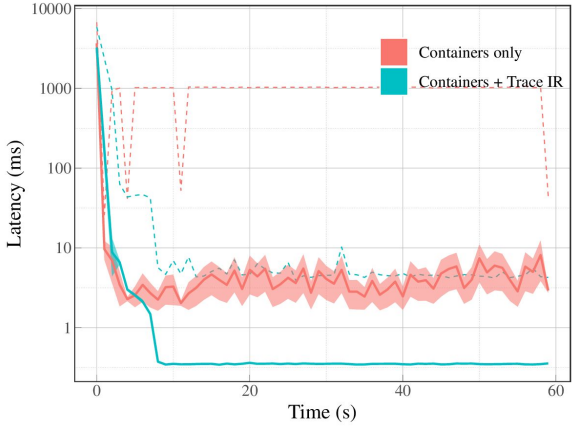
banking



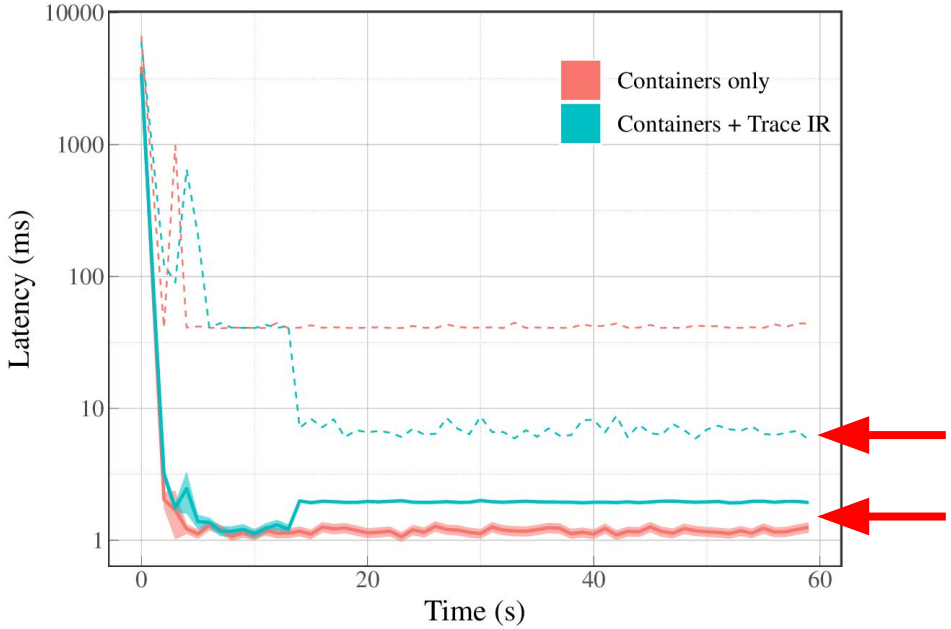
maze



status



upload

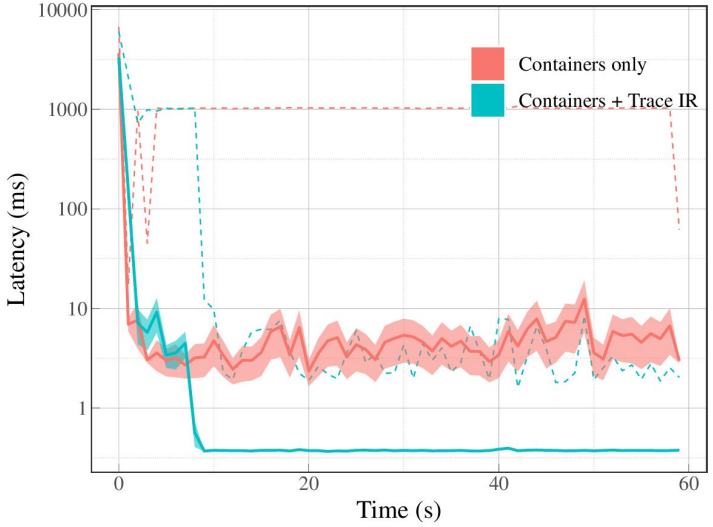


maze

JavaScript’s array versus Rust’s Vec

Summary

- Serverless function accelerators can better performance without burdening programmers
- Language-based isolation achieves better performance, but must be combined with other safety measures
- Containerless uses trace-based compilation to compile JavaScript to Rust



authorize