Wasm/k: Delimited Continuations for WebAssembly

DLS 2020

Donald Pinckney (Northeastern),
Arjun Guha (Northeastern),
Yuriy Brun (UMass Amherst)

$$\begin{tikzcd}
C \arrow[dr] & & \\
& JS \arrow[r] & x86 \\
Go \arrow[ur] & &
\end{tikzcd}$$

+ Language preference
+ Code reuse

$$\begin{tikzcd}
\text{C} \arrow[dr] & & \\
& \text{Wasm} \arrow[r] & \text{x86} \\
\text{Go} \arrow[ur] & &
\end{tikzcd}$$

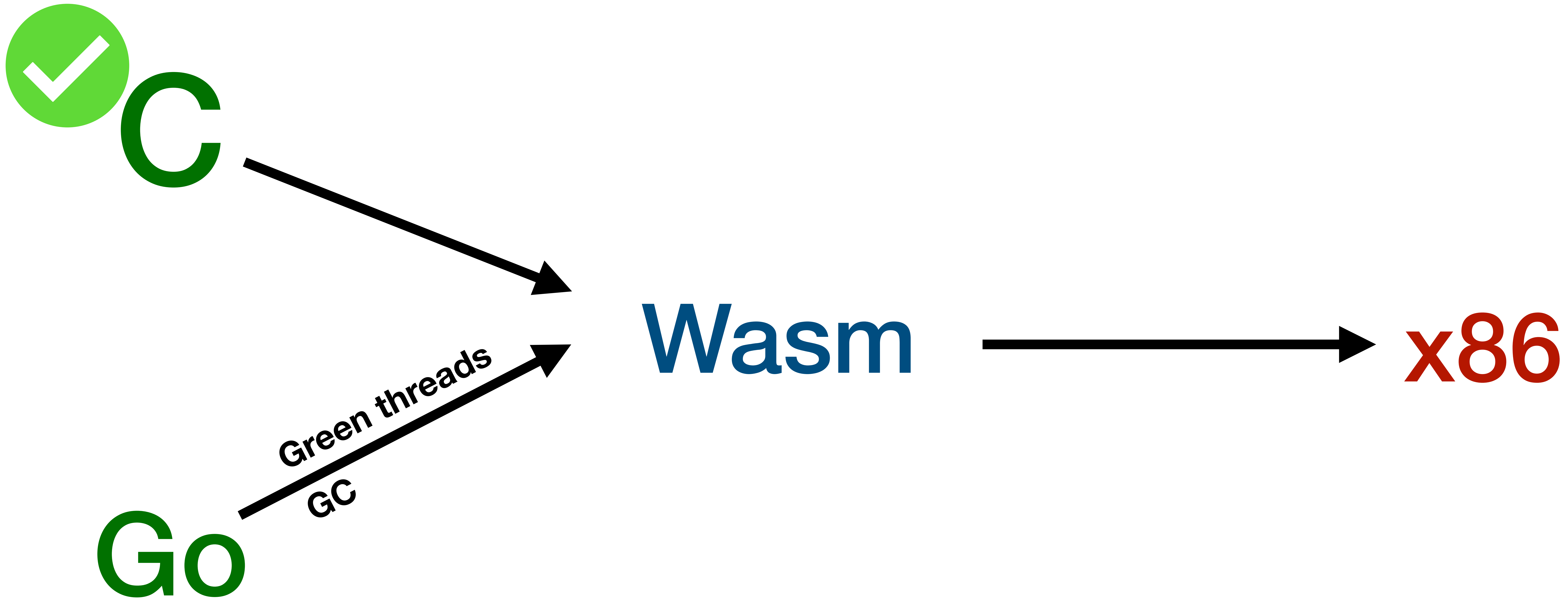
+ Language preference
+ Code reuse

? Fast & consistent performance
? Small code size

$$\begin{tikzcd}
\checkmark\ \text{C} \arrow[dr] & & \\
 & \text{Wasm} \arrow[r] & \text{x86} \\
\text{Go} \arrow[ur] & &
\end{tikzcd}$$

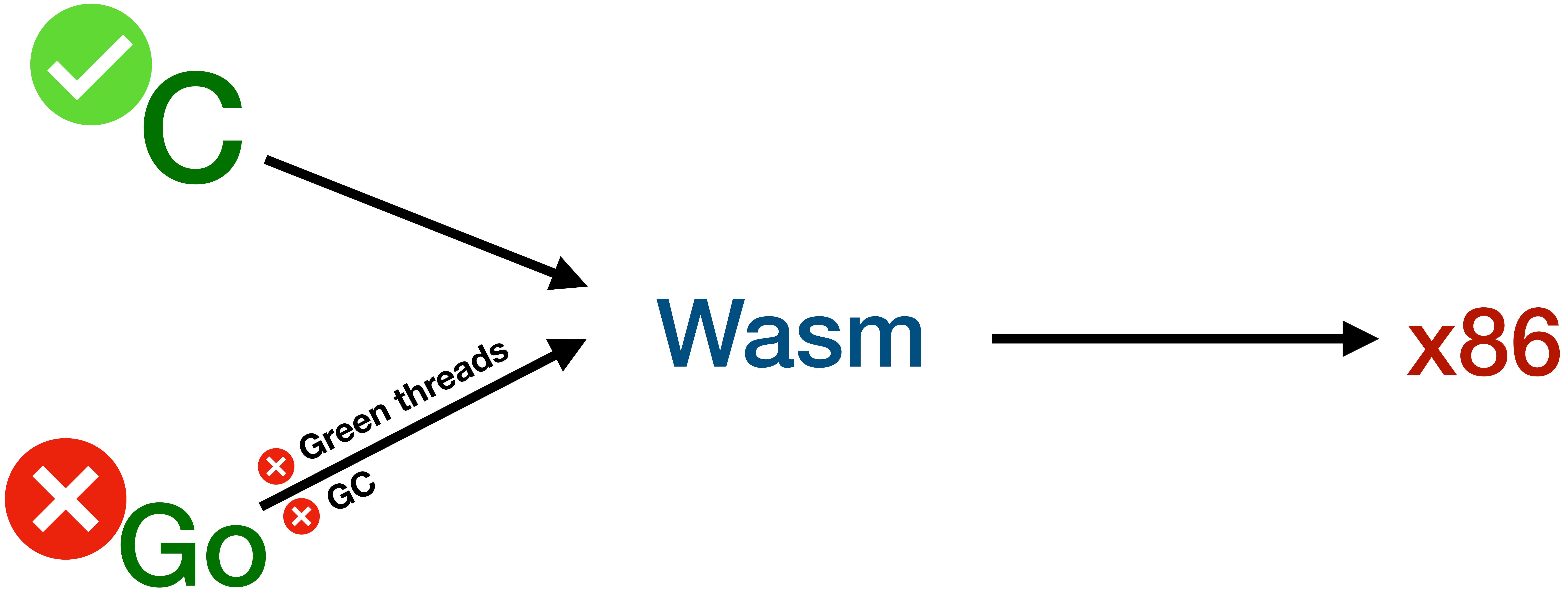
+ Language preference
+ Code reuse

? Fast & consistent performance
? Small code size



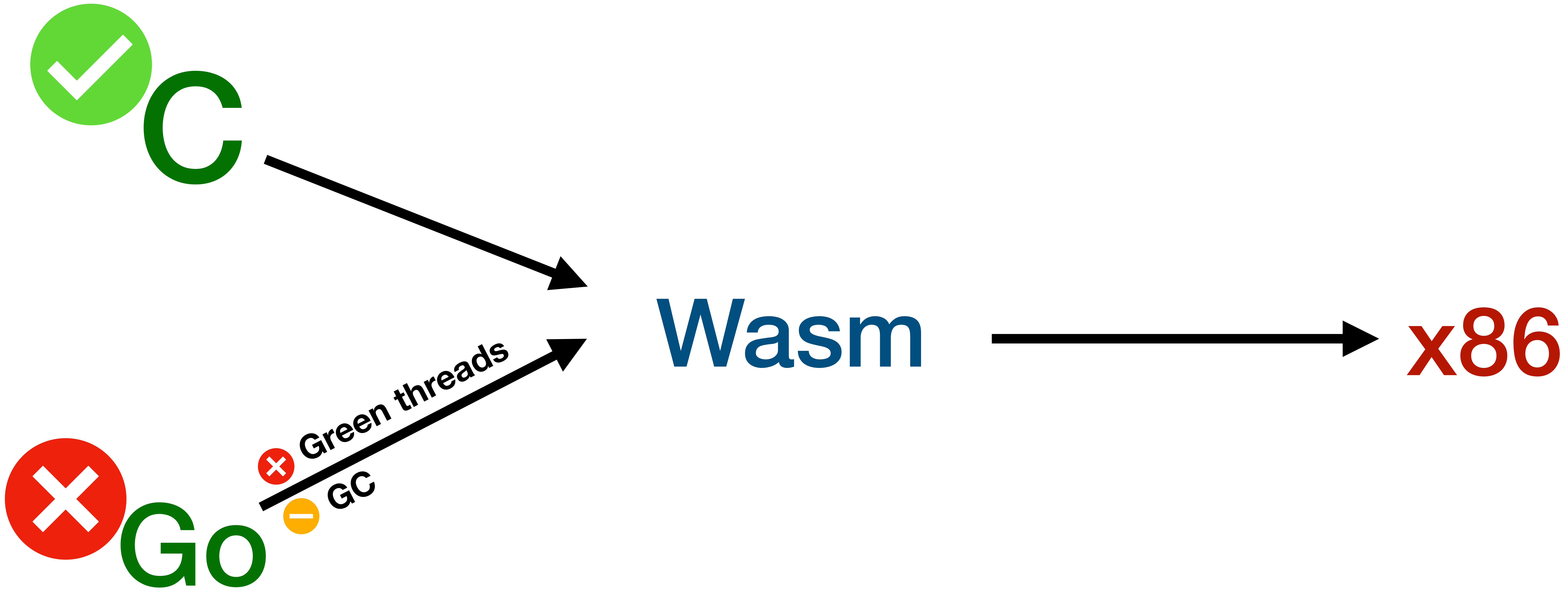
+Language preference
+Code reuse

?Fast & consistent performance
?Small code size



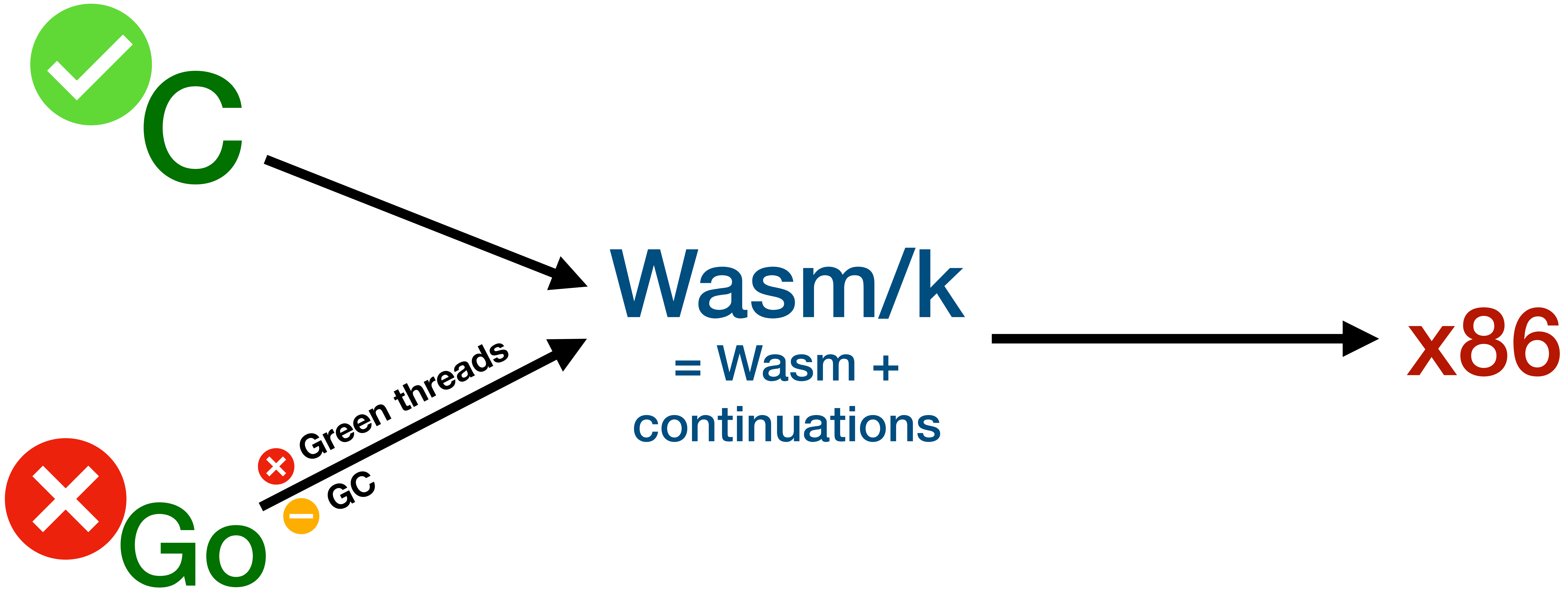
+ Language preference
+ Code reuse

? Fast & consistent performance
? Small code size



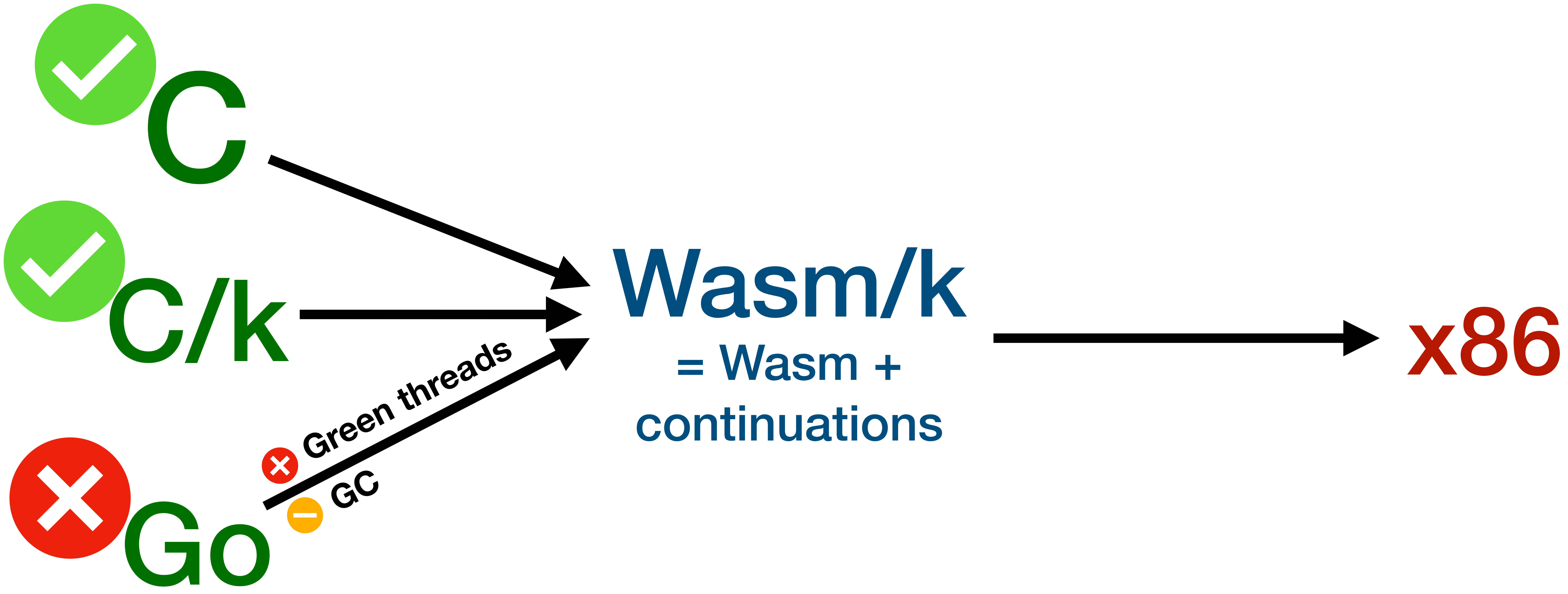
+ Language preference
+ Code reuse

? Fast & consistent performance
? Small code size



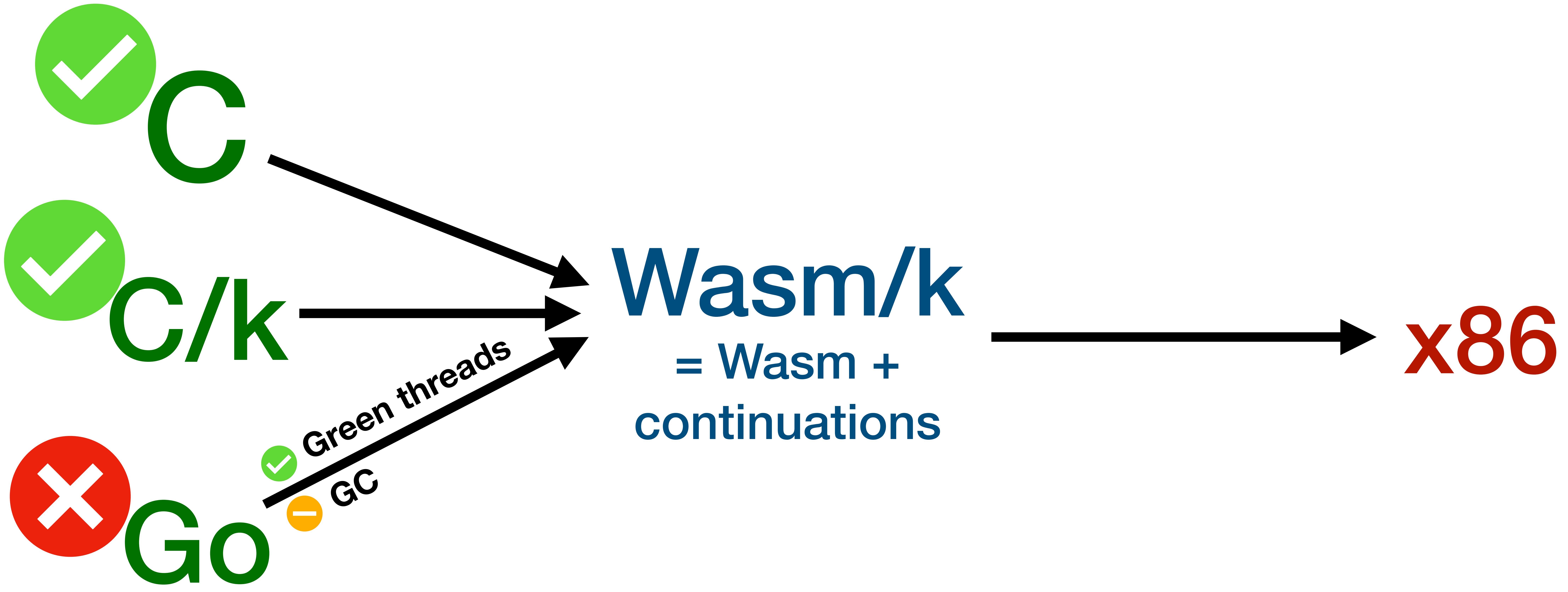
+ Language preference
+ Code reuse

? Fast & consistent performance
? Small code size



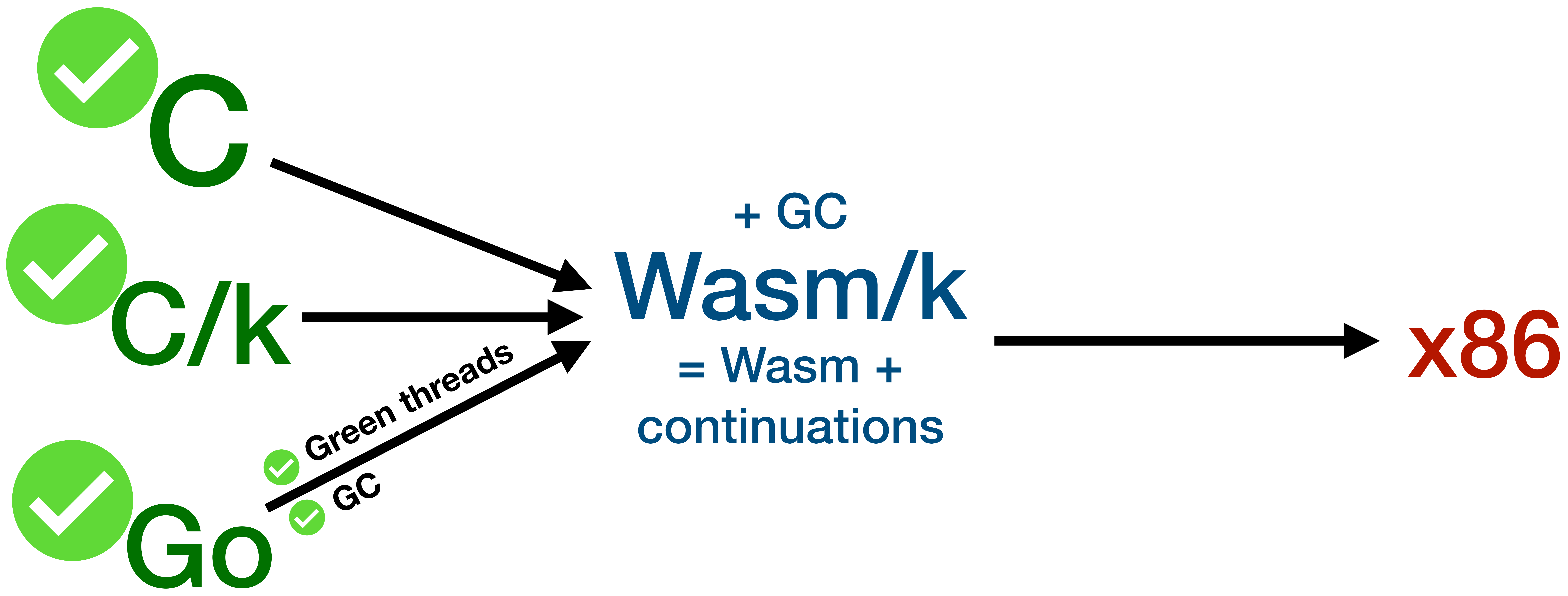
- + Language preference
- + Code reuse

- ? Fast & consistent performance
- ? Small code size



+ Language preference
+ Code reuse

? Fast & consistent performance
? Small code size



+ Language preference
+ Code reuse

? Fast & consistent performance
? Small code size

C $\longrightarrow$ Wasm

Expressions

Formal Stack

```
(5+3)*2
```

```
i32.const 5
i32.const 3
   i32.add
i32.const 2
   i32.mul
```

C $\longrightarrow$ Wasm

C

Functions +
Locals

```c
int f(int x) {
  int y = x;
  return g(y);
}
```

Wasm

Functions +
Locals

```wasm
(func $f (param $x i32)
         (local $y i32)
         (result i32)
  local.get $x
  local.set $y
  local.get $y
  call $g
)
```

$$\begin{tikzcd}\text{C} \arrow[r] & \text{Wasm} \arrow[r] & \text{x86}\end{tikzcd}$$

Expressions

Locals

Function Calls

malloc

Formal Stack

Locals

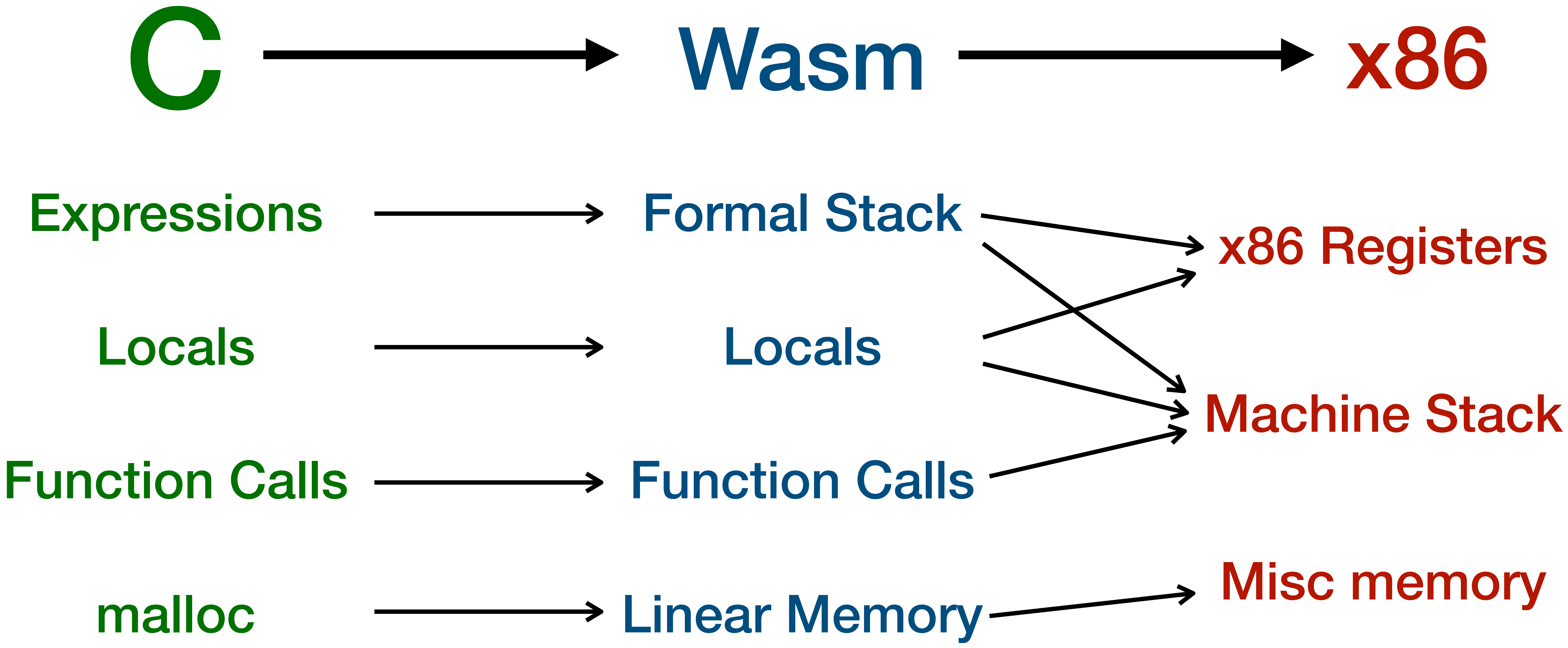
Function Calls

x86 Registers

Machine Stack

$$\begin{tikzcd}
\text{C} \arrow[r] & \text{Wasm} \arrow[r] & \text{x86} \\
\text{Expressions} \arrow[r] & \text{Formal Stack} & \text{x86 Registers} \\
\text{Locals} \arrow[r] & \text{Locals} & \text{Machine Stack} \\
\text{Function Calls} \arrow[r] & \text{Function Calls} & \\
\text{malloc} & &
\end{tikzcd}$$

$$\begin{tikzcd}
\text{C} \arrow[r] & \text{Wasm} \arrow[r] & \text{x86} \\
\text{Expressions} \arrow[r] & \text{Formal Stack} \arrow[dr] \arrow[dddr] & \\
 & & \text{x86 Registers} \\
\text{Locals} \arrow[r] & \text{Locals} \arrow[ur] \arrow[dr] & \\
 & & \text{Machine Stack} \\
\text{Function Calls} \arrow[r] & \text{Function Calls} \arrow[ur] & \\
\text{malloc} & &
\end{tikzcd}$$


C
Wasm
x86
Expressions
Formal Stack
x86 Registers
Locals
Locals
Machine Stack
Function Calls
Function Calls
malloc
Linear Memory
Misc memory

C $\longrightarrow$ Wasm

Pointer to local

```
void f() {
  int x;
  g(&x);
}
```

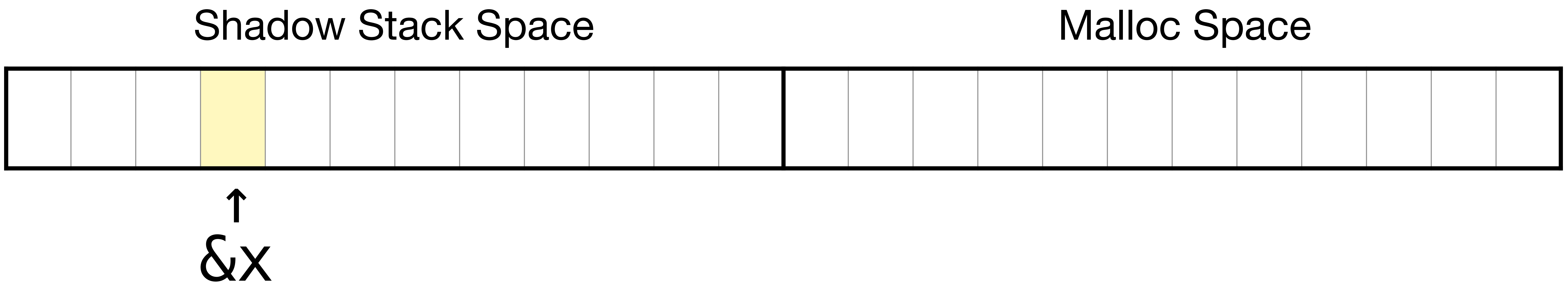
$$\text{C} \longrightarrow \text{Wasm}$$

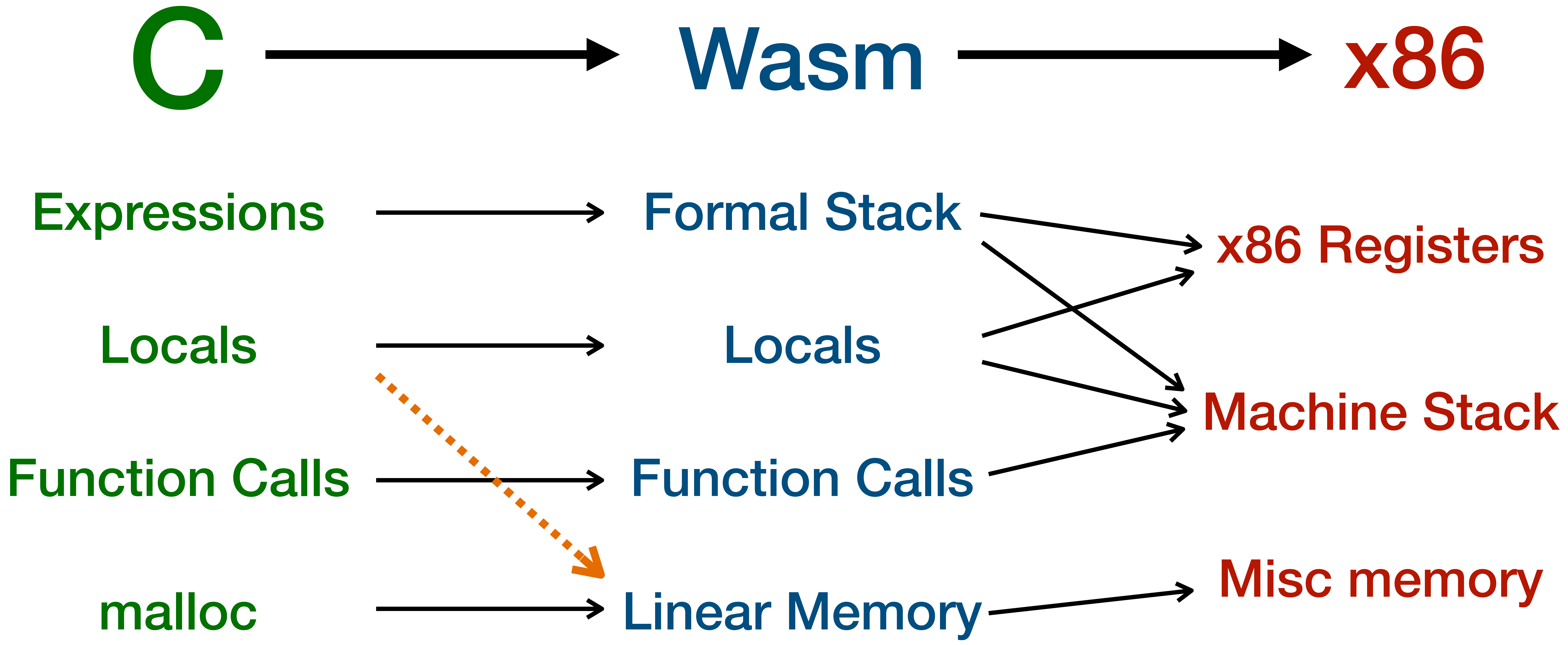
Pointer to local

```
void f() {
  int x;
  g(&x);
}
```

Local in Linear Memory

```
i32.const 4
call $shadow_stack_alloc
call $g
…
i32.const 4
call $shadow_stack_pop
```




C
Wasm
x86
Expressions
Formal Stack
x86 Registers
Locals
Locals
Machine Stack
Function Calls
Function Calls
malloc
Linear Memory
Misc memory

Intro to WebAssembly

The Toolchain + Compilation

$$\begin{tikzcd}
\text{C} \arrow[r] & \text{Wasm} \arrow[r] & \text{x86}
\end{tikzcd}$$

Intro to WebAssembly

The Toolchain + Compilation

$$\begin{tikzcd}
\cancel{C} \arrow[r] & \text{Wasm} \arrow[r] & \text{x86}
\end{tikzcd}$$

Intro to WebAssembly

The Toolchain + Compilation

$$\begin{tikzcd}
\text{Go} \arrow[r] & \text{Wasm} \arrow[r] & \text{x86}
\end{tikzcd}$$

Go → Wasm

```go
x := 0
for {

  x += 1

  f(x);

}
```

```wasm
local.get $x
i32.const 1
i32.add
local.set $x

local.get $x
call $f
```

Go ⟶ Wasm

```
x := 0
for {

  x += 1

  f(x);

}
```

← Might switch Goroutines / green threads!

```
local.get $x
i32.const 1
i32.add
local.set $x
```

```
local.get $x
call $f
```

Go $\longrightarrow$ Wasm

```go
x := 0
for {

  x += 1

  f(x);

}
```

← Might switch Goroutines / green threads!

~~local.get $x~~
~~i32.const 1~~
~~i32.add~~
~~local.set $x~~

~~local.get $x~~
~~call $f~~

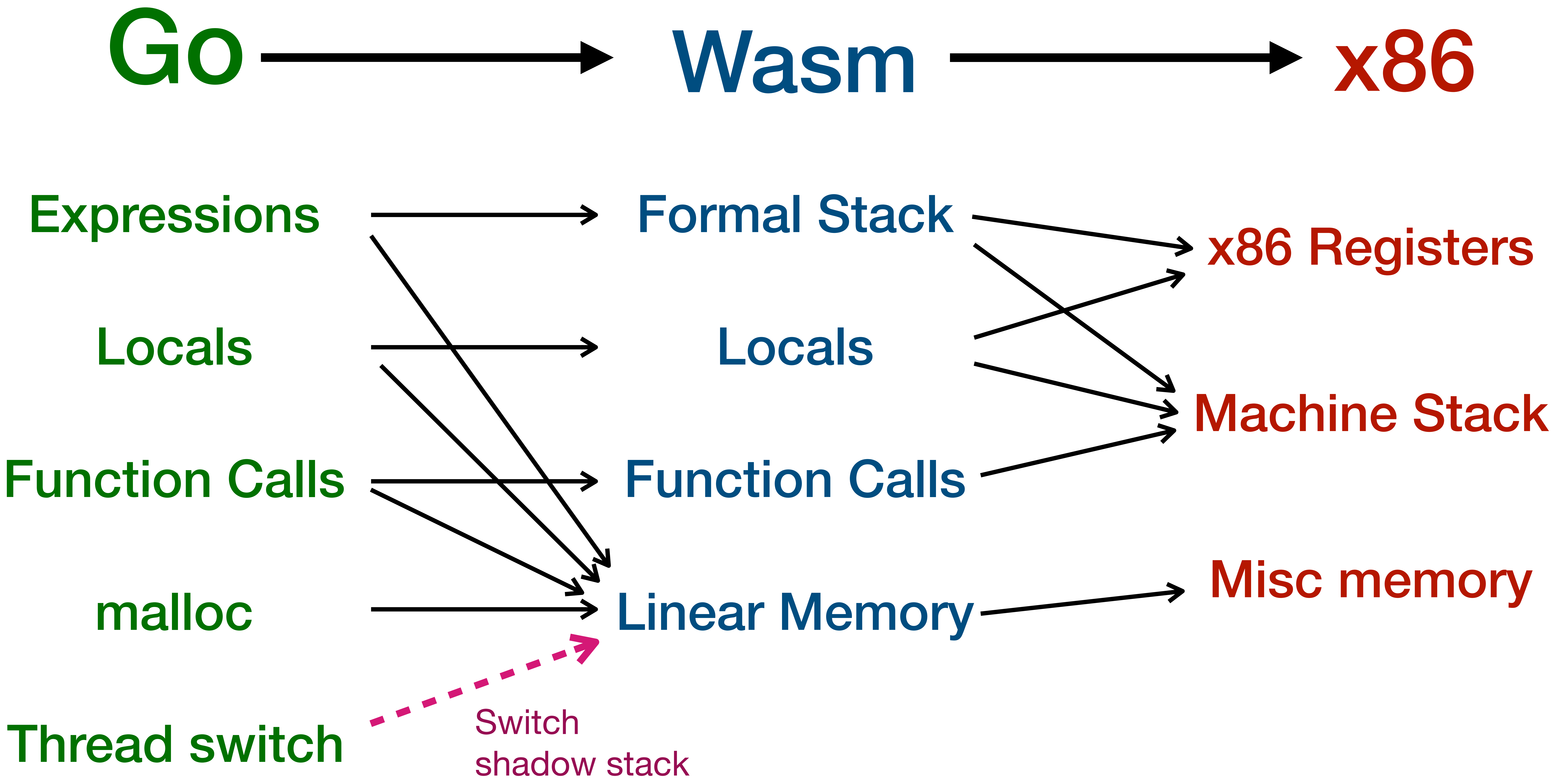
Go ⟶ Wasm

```go
x := 0
for {

  x += 1

  f(x);

}
```

← Might switch Goroutines / green threads!

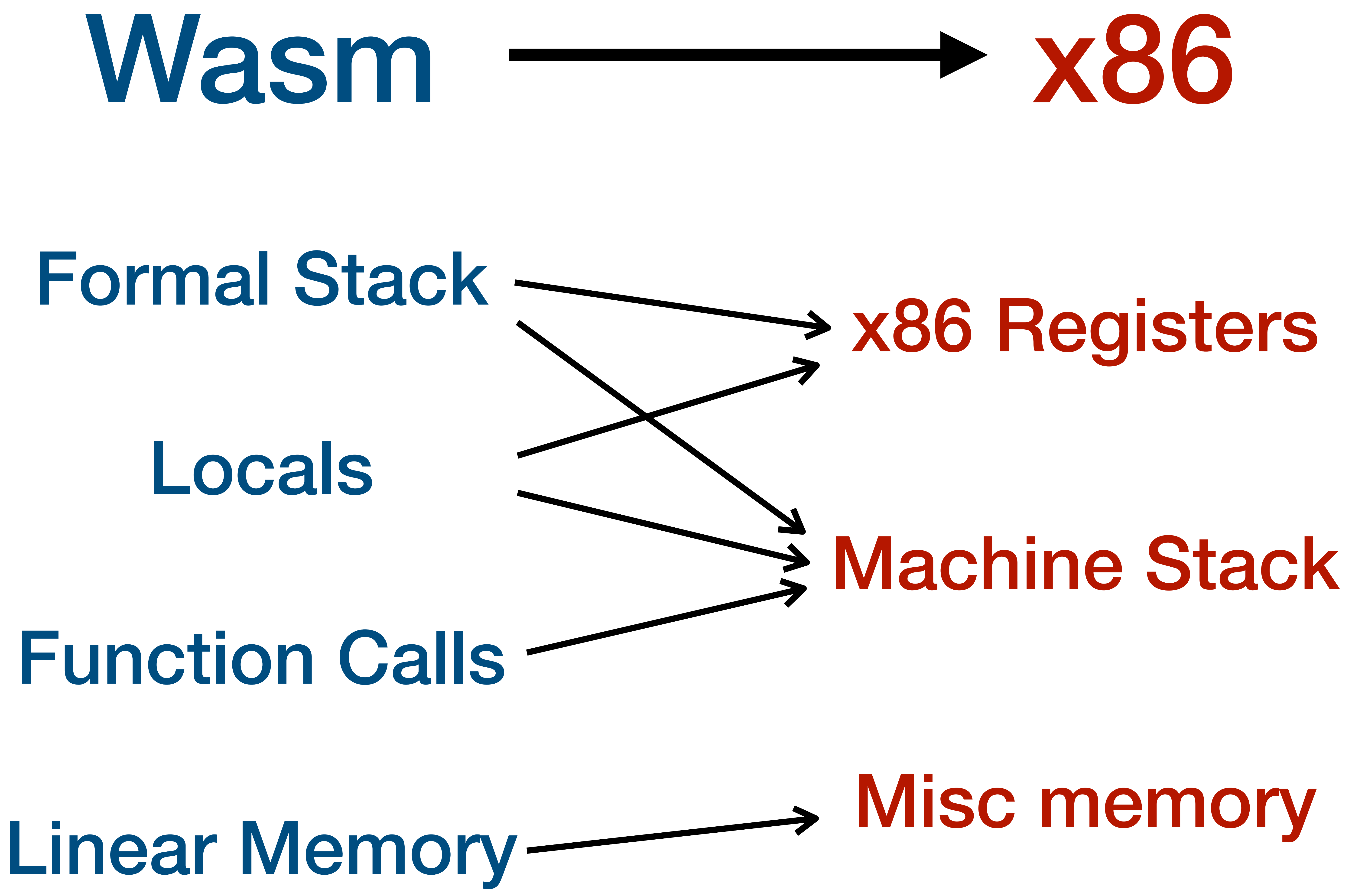
```wasm
local.get $x
i32.const 1
i32.add
local.set $x
```

```wasm
local.get $x
call $push_shadow_stack
local.get $x
call $f
call $pop_shadow_stack
local.set $x
```

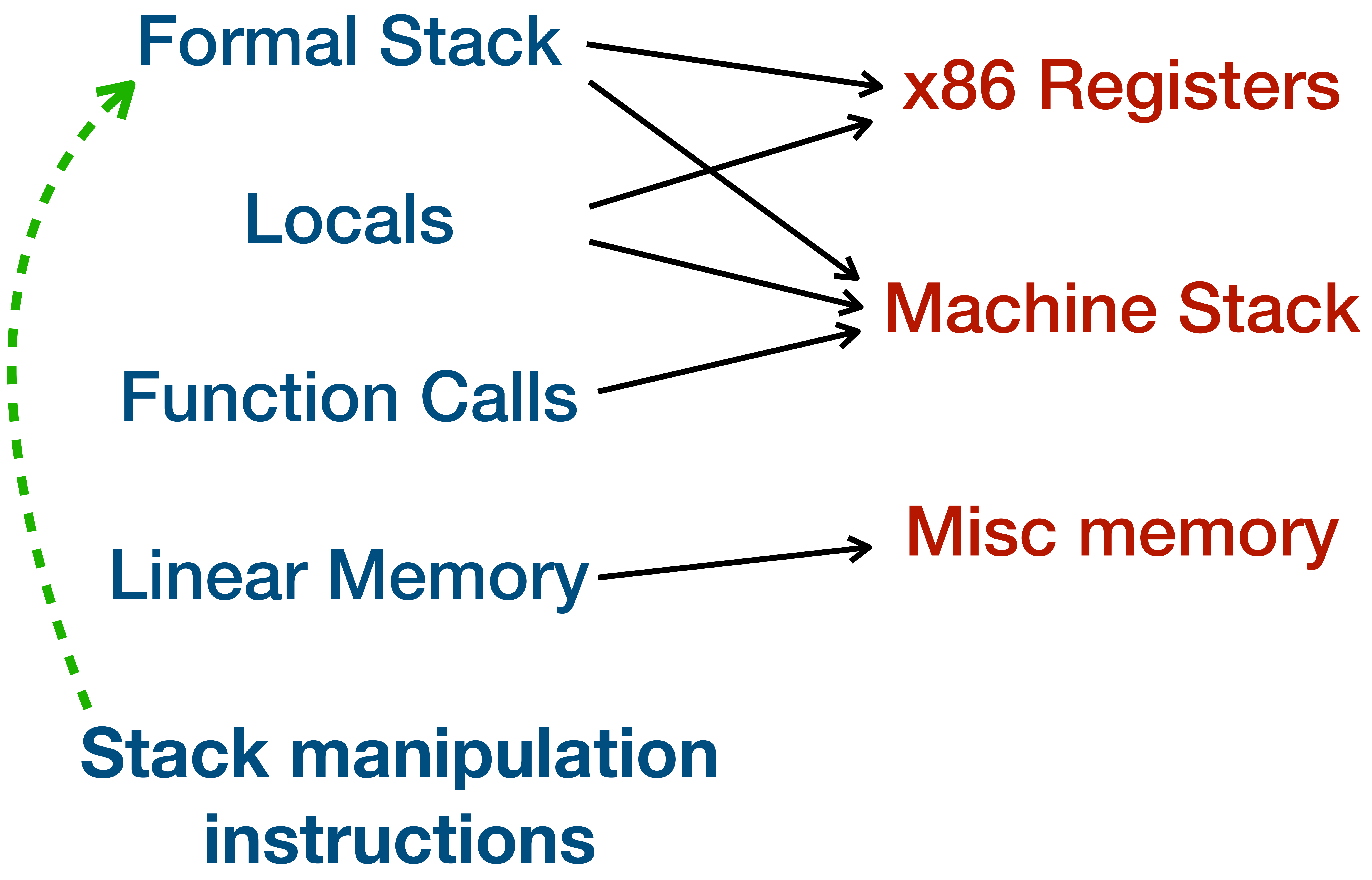

Go
Wasm
x86
Expressions
Locals
Function Calls
malloc
Thread switch
Formal Stack
Locals
Function Calls
Linear Memory
x86 Registers
Machine Stack
Misc memory
Switch
shadow stack

Key Problem:

Go needs to manipulate the stack, but WebAssembly disallows manipulating the formal stack


Wasm
x86
Formal Stack
Locals
Function Calls
Linear Memory
x86 Registers
Machine Stack
Misc memory

Wasm/k $\longrightarrow$ x86



$$\begin{tikzcd}
\text{C/k} \arrow[r] & \text{Wasm/k} \arrow[r] & \text{x86} \\
\text{Expressions} \arrow[r] & \text{Formal Stack} \arrow[r] \arrow[dr] & \text{x86 Registers} \\
\text{Locals} \arrow[r] \arrow[ddr, dotted] & \text{Locals} \arrow[ur] \arrow[r] & \text{Machine Stack} \\
\text{Function Calls} \arrow[r] & \text{Function Calls} \arrow[ur] & \\
\text{malloc} \arrow[r] & \text{Linear Memory} \arrow[r] & \text{Misc memory} \\
\text{Stack manipulation functions} \arrow[r] & \text{Stack manipulation instructions} \arrow[uuuu, dashed, bend left] &
\end{tikzcd}$$

C/k = C +		**Wasm/k** = Wasm +
`control()`	→	`control`
`restore()`	→	`restore`
`continuation_copy()`	→	`continuation_copy`
`continuation_delete()`	→	`continuation_delete`
`prompt()`	→	`prompt … end`

Programming in C/k

Green threads: `control()` and `restore()`

```
std::vector<uint64_t> Q;

uint64_t dequeue() {
    uint64_t next_k = Q.back(); Q.pop_back();
    return next_k;
}

void yield_handler(uint64_t k, uint64_t arg) {
    Q.insert(Q.begin(), k);
    restore(dequeue(), 0);
}
void thread_yield() {
    control(yield_handler, 0);
}
```

```
void thread_main() {
    std::cout << "A" << std::endl;
    thread_yield();
    std::cout << "B" << std::endl;
}
int main() {
    thread_create(thread_main);
    thread_create(thread_main);
    join_all_threads();
}
```

Programming in C/k

Green threads: `control()` and `restore()`

```
std::vector<uint64_t> Q;

uint64_t dequeue() {
    uint64_t next_k = Q.back(); Q.pop_back();
    return next_k;
}

void yield_handler(uint64_t k, uint64_t arg) {
    Q.insert(Q.begin(), k);
    restore(dequeue(), 0);
}
void thread_yield() {
    control(yield_handler, 0);
}
```

```
void thread_main() {
    std::cout << "A" << std::endl;
    thread_yield();
    std::cout << "B" << std::endl;
}
int main() {
    thread_create(thread_main);
    thread_create(thread_main);
    join_all_threads();
}
```

Programming in C/k

Green threads: `control()` and `restore()`

```
std::vector<uint64_t> Q;

uint64_t dequeue() {
    uint64_t next_k = Q.back(); Q.pop_back();
    return next_k;
}

void yield_handler(uint64_t k, uint64_t arg) {
    Q.insert(Q.begin(), k);
    restore(dequeue(), 0);
}
void thread_yield() {
    control(yield_handler, 0);
}
```

```
void thread_main() {
    std::cout << "A" << std::endl;
    thread_yield();
    std::cout << "B" << std::endl;
}
int main() {
    thread_create(thread_main);
    thread_create(thread_main);
    join_all_threads();
}
```

Programming in C/k

Green threads: `control()` and `restore()`

```
std::vector<uint64_t> Q;

uint64_t dequeue() {
    uint64_t next_k = Q.back(); Q.pop_back();
    return next_k;
}

void yield_handler(uint64_t k, uint64_t arg) {
    Q.insert(Q.begin(), k);
    restore(dequeue(), 0);
}
void thread_yield() {
    control(yield_handler, 0);
}
```

```
void thread_main() {
    std::cout << "A" << std::endl;
    thread_yield();
    std::cout << "B" << std::endl;
}
int main() {
    thread_create(thread_main);
    thread_create(thread_main);
    join_all_threads();
}
```

Programming in C/k

Green threads: `control()` and `restore()`

```
std::vector<uint64_t> Q;

uint64_t dequeue() {
    uint64_t next_k = Q.back(); Q.pop_back();
    return next_k;
}

void yield_handler(uint64_t k, uint64_t arg) {
    Q.insert(Q.begin(), k);
    restore(dequeue(), 0);
}
void thread_yield() {
    control(yield_handler, 0);
}
```

```
void thread_main() {
    std::cout << "A" << std::endl;
    thread_yield();
    std::cout << "B" << std::endl;
}
int main() {
    thread_create(thread_main);
    thread_create(thread_main);
    join_all_threads();
}
```

Programming in C/k

Generators: `continuation_delete()`

```c
void yield_handler(k_id k, Generator *g) {
    g->after_yield = k;
    restore(g->after_next, g->value);
}
void gen_yield(uint64_t v, Generator *g) {
    g->value = v;
    control(yield_handler, g);
}
// Next implementation
void next_handler(k_id k, Generator *g) {
    g->after_next = k;
    restore(g->after_yield, 0);
}
uint64_t gen_next(Generator *g) {
    return control(next_handler, g);
}
// Freeing a generator
void free_generator(Generator *g) {
    continuation_delete(g->after_yield); free(g);
}
```

```c
void example_generator(Generator *g) {
    uint64_t i = 0;
    while(1) { gen_yield(i++, g); }
}
int main() {
    Generator *g = make_generator(example_generator);
    for(int i = 0; i < 10; i++)
        printf("%llu\n", gen_next(g));
    free_generator(g);
    return 0;
}
```

Programming in C/k

Generators: `continuation_delete()`

```c
void yield_handler(k_id k, Generator *g) {
    g->after_yield = k;
    restore(g->after_next, g->value);
}
void gen_yield(uint64_t v, Generator *g) {
    g->value = v;
    control(yield_handler, g);
}
// Next implementation
void next_handler(k_id k, Generator *g) {
    g->after_next = k;
    restore(g->after_yield, 0);
}
uint64_t gen_next(Generator *g) {
    return control(next_handler, g);
}
// Freeing a generator
void free_generator(Generator *g) {
    continuation_delete(g->after_yield); free(g);
}
```

```c
void example_generator(Generator *g) {
    uint64_t i = 0;
    while(1) { gen_yield(i++, g); }
}
int main() {
    Generator *g = make_generator(example_generator);
    for(int i = 0; i < 10; i++)
        printf("%llu\n", gen_next(g));
    free_generator(g);
    return 0;
}
```

Programming in C/k

Generators: `continuation_delete()`

```
void yield_handler(k_id k, Generator *g) {
    g->after_yield = k;
    restore(g->after_next, g->value);
}
void gen_yield(uint64_t v, Generator *g) {
    g->value = v;
    control(yield_handler, g);
}
// Next implementation
void next_handler(k_id k, Generator *g) {
    g->after_next = k;
    restore(g->after_yield, 0);
}
uint64_t gen_next(Generator *g) {
    return control(next_handler, g);
}
// Freeing a generator
void free_generator(Generator *g) {
    continuation_delete(g->after_yield); free(g);
}
```

```
void example_generator(Generator *g) {
    uint64_t i = 0;
    while(1) { gen_yield(i++, g); }
}
int main() {
    Generator *g = make_generator(example_generator);
    for(int i = 0; i < 10; i++)
        printf("%llu\n", gen_next(g));
    free_generator(g);
    return 0;
}
```

Programming in C/k

Generators: `continuation_delete()`

```
void yield_handler(k_id k, Generator *g) {
    g->after_yield = k;
    restore(g->after_next, g->value);
}
void gen_yield(uint64_t v, Generator *g) {
    g->value = v;
    control(yield_handler, g);
}
// Next implementation
void next_handler(k_id k, Generator *g) {
    g->after_next = k;
    restore(g->after_yield, 0);
}
uint64_t gen_next(Generator *g) {
    return control(next_handler, g);
}
// Freeing a generator
void free_generator(Generator *g) {
    continuation_delete(g->after_yield); free(g);
}
```

```
void example_generator(Generator *g) {
    uint64_t i = 0;
    while(1) { gen_yield(i++, g); }
}
int main() {
    Generator *g = make_generator(example_generator);
    for(int i = 0; i < 10; i++)
        printf("%llu\n", gen_next(g));
    free_generator(g);
    return 0;
}
```

Programming in C/k

Generators: `continuation_delete()`

```
void yield_handler(k_id k, Generator *g) {
    g->after_yield = k;
    restore(g->after_next, g->value);
}
void gen_yield(uint64_t v, Generator *g) {
    g->value = v;
    control(yield_handler, g);
}
// Next implementation
void next_handler(k_id k, Generator *g) {
    g->after_next = k;
    restore(g->after_yield, 0);
}
uint64_t gen_next(Generator *g) {
    return control(next_handler, g);
}
// Freeing a generator
void free_generator(Generator *g) {
    continuation_delete(g->after_yield); free(g);
}
```

```
void example_generator(Generator *g) {
    uint64_t i = 0;
    while(1) { gen_yield(i++, g); }
}
int main() {
    Generator *g = make_generator(example_generator);
    for(int i = 0; i < 10; i++)
        printf("%llu\n", gen_next(g));
    free_generator(g);
    return 0;
}
```

Programming in C/k

Generators: `continuation_delete()`

```
void yield_handler(k_id k, Generator *g) {
    g->after_yield = k;
    restore(g->after_next, g->value);
}
void gen_yield(uint64_t v, Generator *g) {
    g->value = v;
    control(yield_handler, g);
}
// Next implementation
void next_handler(k_id k, Generator *g) {
    g->after_next = k;
    restore(g->after_yield, 0);
}
uint64_t gen_next(Generator *g) {
    return control(next_handler, g);
}
// Freeing a generator
void free_generator(Generator *g) {
    continuation_delete(g->after_yield); free(g);
}
```

```
void example_generator(Generator *g) {
    uint64_t i = 0;
    while(1) { gen_yield(i++, g); }
}
int main() {
    Generator *g = make_generator(example_generator);
    for(int i = 0; i < 10; i++)
        printf("%llu\n", gen_next(g));
    free_generator(g);
    return 0;
}
```

Programming in C/k

Probabilistic Programming: `continuation_copy()`

```
struct ContinuationThunk {
    k_id continuation; // The continuation to resume
    uint64_t value; // The value to pass to the continuation
};
// vector of thunks which need to be executed
std::vector<ContinuationThunk *> to_execute;

std::map<uint64_t, double> *driver(uint64_t (*body)()) {
    auto *results = new std::vector<uint64_t>();
    results->push_back(body());
    if(rest.size() > 0) {
        ContinuationThunk *t = rest.back(); rest.pop_back();
        restore(t->continuation, t->value);
    }
    return count_probs(results);
}

void uniform_handler(k_id k, std::vector<uint64_t> *args) {
    for(auto it = std::next(args->begin());
        it != args->end(); ++it) {
        to_execute.push_back(new ContinuationThunk {
            .continuation=continuation_copy(k),
            .value=*it});
    }
    restore(k, args[0]);
}
uint64_t uniform(std::vector<uint64_t> *args) {
    return control(uniform_handler, args);
}
```

```
uint64_t sum_d6() {
    auto *d6 = new std::vector<uint64_t> {1, 2, 3, 4, 5, 6};
    return uniform(d6) + uniform(d6);
}
int main() {
    std::cout << *driver(sum_d6) << std::endl; return 0;
}
```

Programming in C/k

Probabilistic Programming: `continuation_copy()`

```
struct ContinuationThunk {
    k_id continuation; // The continuation to resume
    uint64_t value; // The value to pass to the continuation
};
// vector of thunks which need to be executed
std::vector<ContinuationThunk *> to_execute;

std::map<uint64_t, double> *driver(uint64_t (*body)()) {
    auto *results = new std::vector<uint64_t>();
    results->push_back(body());
    if(rest.size() > 0) {
        ContinuationThunk *t = rest.back(); rest.pop_back();
        restore(t->continuation, t->value);
    }
    return count_probs(results);
}

void uniform_handler(k_id k, std::vector<uint64_t> *args) {
    for(auto it = std::next(args->begin());
        it != args->end(); ++it) {
        to_execute.push_back(new ContinuationThunk {
            .continuation=continuation_copy(k),
            .value=*it});
    }
    restore(k, args[0]);
}
uint64_t uniform(std::vector<uint64_t> *args) {
    return control(uniform_handler, args);
}
```

```
uint64_t sum_d6() {
    auto *d6 = new std::vector<uint64_t> {1, 2, 3, 4, 5, 6};
    return uniform(d6) + uniform(d6);
}
int main() {
    std::cout << *driver(sum_d6) << std::endl; return 0;
}
```

Programming in C/k

Probabilistic Programming: `continuation_copy()`

```
struct ContinuationThunk {
    k_id continuation; // The continuation to resume
    uint64_t value; // The value to pass to the continuation
};
// vector of thunks which need to be executed
std::vector<ContinuationThunk *> to_execute;

std::map<uint64_t, double> *driver(uint64_t (*body)()) {
    auto *results = new std::vector<uint64_t>();
    results->push_back(body());
    if(rest.size() > 0) {
        ContinuationThunk *t = rest.back(); rest.pop_back();
        restore(t->continuation, t->value);
    }
    return count_probs(results);
}

void uniform_handler(k_id k, std::vector<uint64_t> *args) {
    for(auto it = std::next(args->begin());
        it != args->end(); ++it) {
        to_execute.push_back(new ContinuationThunk {
            .continuation=continuation_copy(k),
            .value=*it});
    }
    restore(k, args[0]);
}
uint64_t uniform(std::vector<uint64_t> *args) {
    return control(uniform_handler, args);
}
```

```
uint64_t sum_d6() {
    auto *d6 = new std::vector<uint64_t> {1, 2, 3, 4, 5, 6};
    return uniform(d6) + uniform(d6);
}
int main() {
    std::cout << *driver(sum_d6) << std::endl; return 0;
}
```

Programming in C/k

Probabilistic Programming: `continuation_copy()`

```
struct ContinuationThunk {
    k_id continuation; // The continuation to resume
    uint64_t value; // The value to pass to the continuation
};
// vector of thunks which need to be executed
std::vector<ContinuationThunk *> to_execute;

std::map<uint64_t, double> *driver(uint64_t (*body)()) {
    auto *results = new std::vector<uint64_t>();
    results->push_back(body());
    if(rest.size() > 0) {
        ContinuationThunk *t = rest.back(); rest.pop_back();
        restore(t->continuation, t->value);
    }
    return count_probs(results);
}

void uniform_handler(k_id k, std::vector<uint64_t> *args) {
    for(auto it = std::next(args->begin());
        it != args->end(); ++it) {
        to_execute.push_back(new ContinuationThunk {
            .continuation=continuation_copy(k),
            .value=*it});
    }
    restore(k, args[0]);
}
uint64_t uniform(std::vector<uint64_t> *args) {
    return control(uniform_handler, args);
}
```

```
uint64_t sum_d6() {
    auto *d6 = new std::vector<uint64_t> {1, 2, 3, 4, 5, 6};
    return uniform(d6) + uniform(d6);
}
int main() {
    std::cout << *driver(sum_d6) << std::endl; return 0;
}
```

Programming in C/k

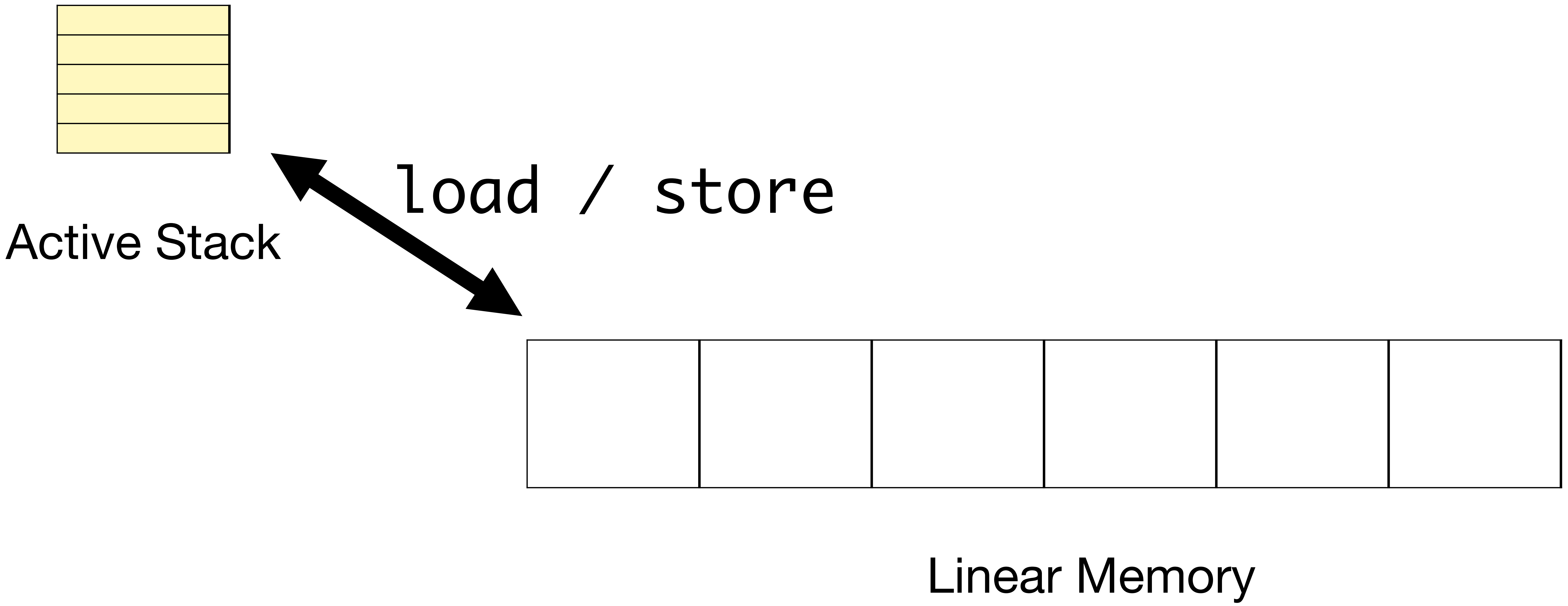
Probabilistic Programming: `continuation_copy()`

```
struct ContinuationThunk {
    k_id continuation; // The continuation to resume
    uint64_t value; // The value to pass to the continuation
};
// vector of thunks which need to be executed
std::vector<ContinuationThunk *> to_execute;

std::map<uint64_t, double> *driver(uint64_t (*body)()) {
    auto *results = new std::vector<uint64_t>();
    results->push_back(body());
    if(rest.size() > 0) {
        ContinuationThunk *t = rest.back(); rest.pop_back();
        restore(t->continuation, t->value);
    }
    return count_probs(results);
}

void uniform_handler(k_id k, std::vector<uint64_t> *args) {
    for(auto it = std::next(args->begin());
        it != args->end(); ++it) {
        to_execute.push_back(new ContinuationThunk {
            .continuation=continuation_copy(k),
            .value=*it});
    }
    restore(k, args[0]);
}
uint64_t uniform(std::vector<uint64_t> *args) {
    return control(uniform_handler, args);
}
```

```
uint64_t sum_d6() {
    auto *d6 = new std::vector<uint64_t> {1, 2, 3, 4, 5, 6};
    return uniform(d6) + uniform(d6);
}
int main() {
    std::cout << *driver(sum_d6) << std::endl; return 0;
}
```

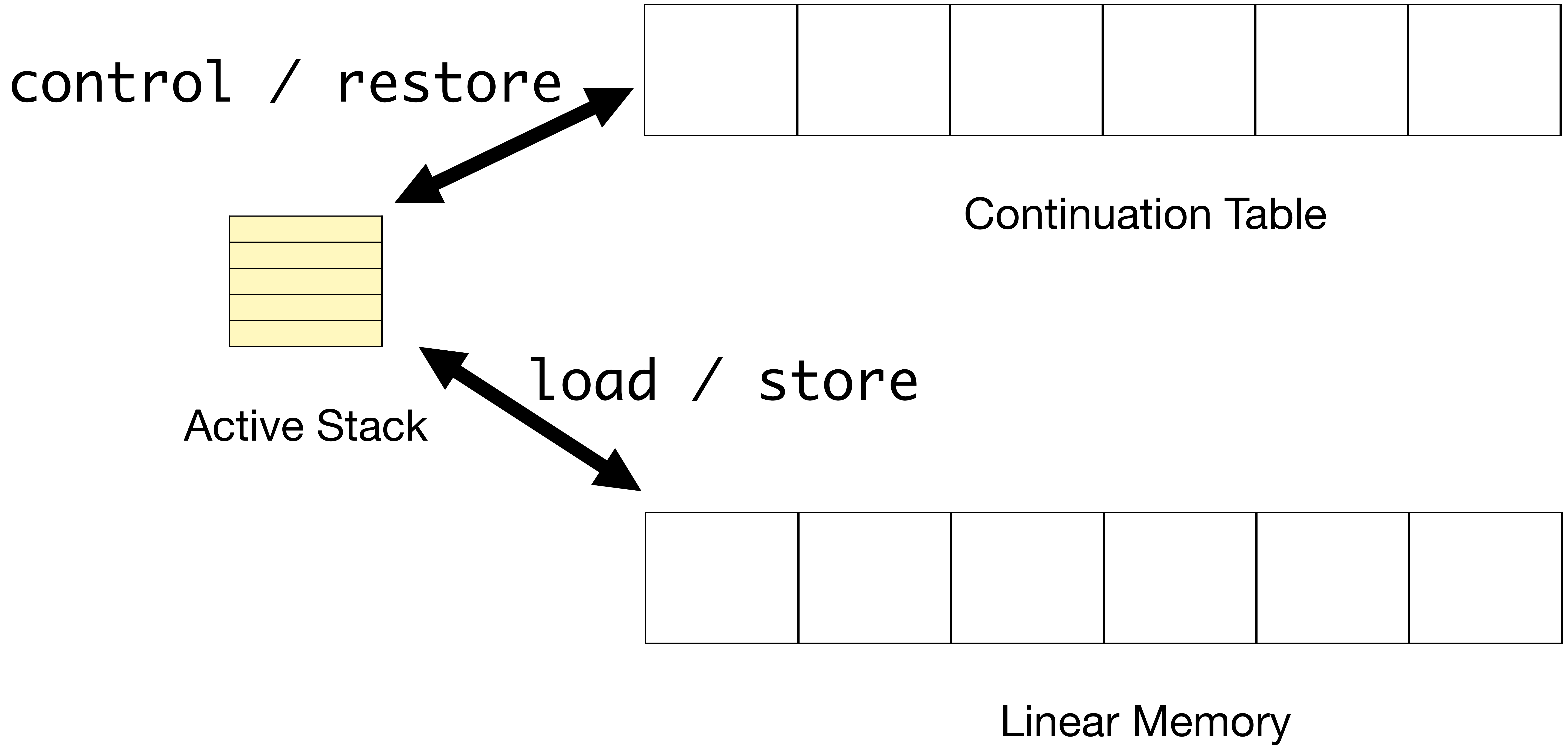
Wasm/k Semantics

The continuation table



Wasm/k Semantics

The continuation table

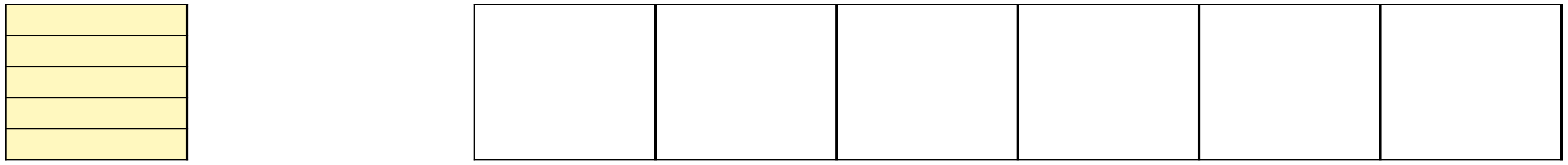


Wasm/k Semantics

`control` and `restore`

$$s;\ v_l^*;\ L^{\max}[(\mathbf{i64.const}\ v)\ (\mathbf{control}\ h)] \rightsquigarrow_i s';\ \epsilon;\ (\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ (\mathbf{call}\ h)\ \mathbf{trap}$$

$$s;\ v_l^*;\ L^{\max}[(\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ \mathbf{restore}] \rightsquigarrow_i s';\ v_l^{*\prime};\ L^{\max\prime}[(\mathbf{i64.const}\ v)]$$



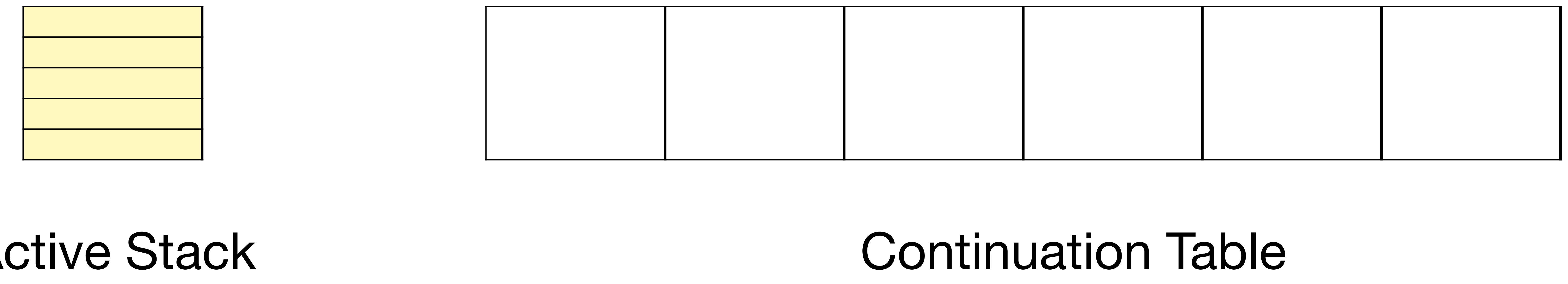
Wasm/k Semantics

`control` and `restore`

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\mathbf{i64.const}\ v)\ (\mathbf{control}\ h)] \leadsto_i s';\ \epsilon;\ (\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ (\mathbf{call}\ h)\ \mathbf{trap}$$

Capture the stack

$$s;\ v_l^*;\ L^{\max}[(\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ \mathbf{restore}] \leadsto_i s';\ v_l^{*\prime};\ L^{\max\prime}[(\mathbf{i64.const}\ v)]$$



Wasm/k Semantics

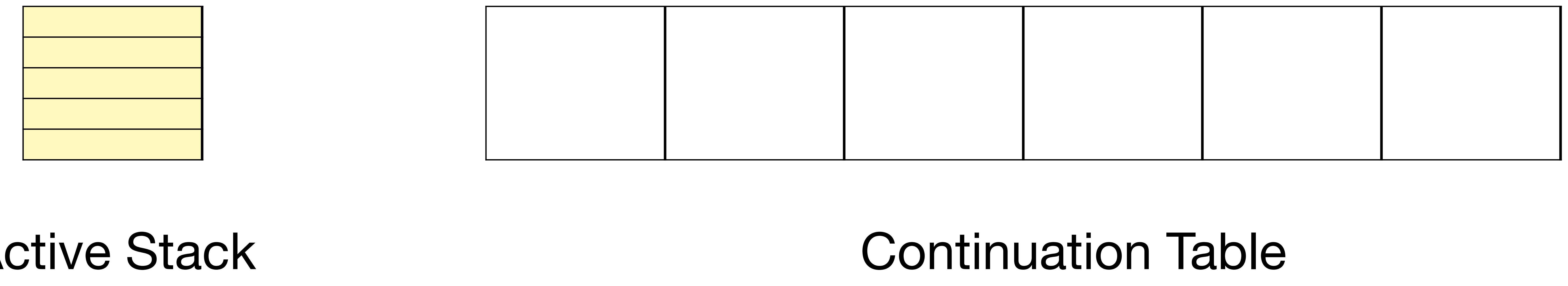
`control` and `restore`

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\textbf{i64.const}\ v)\ (\textbf{control}\ h)] \rightsquigarrow_i \boxed{s';}\ \epsilon;\ (\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ (\textbf{call}\ h)\ \textbf{trap}$$

Capture the stack

where s’ = s but with context L^{max} and v_l^* stored at index k

$$s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ \textbf{restore}] \rightsquigarrow_i s';\ v_l^{*\prime};\ L^{\max\prime}[(\textbf{i64.const}\ v)]$$



Wasm/k Semantics

`control` and `restore`

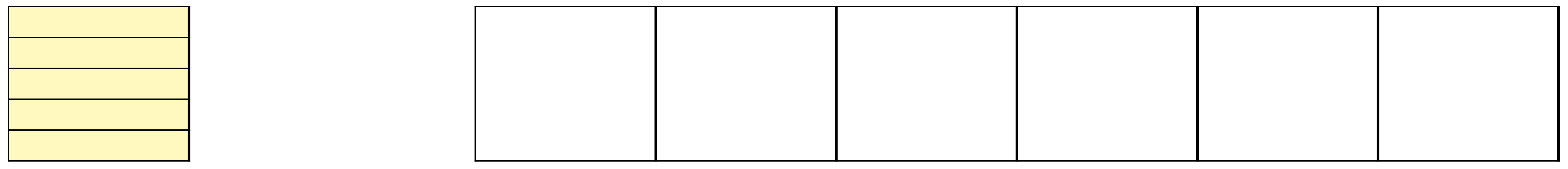
$$s;\ v_l^*;\ \boxed{L^{\max}}[(\mathbf{i64.const}\ v)\ (\mathbf{control}\ h)] \rightsquigarrow_i \boxed{s';}\ \boxed{\epsilon;\ (\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ (\mathbf{call}\ h)}\ \mathbf{trap}$$

Capture the stack

where s' = s but with context L^{max} and v_l^* stored at index k

Execute h in fresh stack.
k is a new ID

$$s;\ v_l^*;\ L^{\max}[(\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ \mathbf{restore}] \rightsquigarrow_i s';\ v_l^{*\prime};\ L^{\max\prime}[(\mathbf{i64.const}\ v)]$$



Active Stack

Continuation Table

Wasm/k Semantics

`control` and `restore`

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\mathbf{i64.const}\ v)\ (\mathbf{control}\ h)] \rightsquigarrow_i \boxed{s';}\ \boxed{\epsilon;\ (\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ (\mathbf{call}\ h)}\ \mathbf{trap}$$

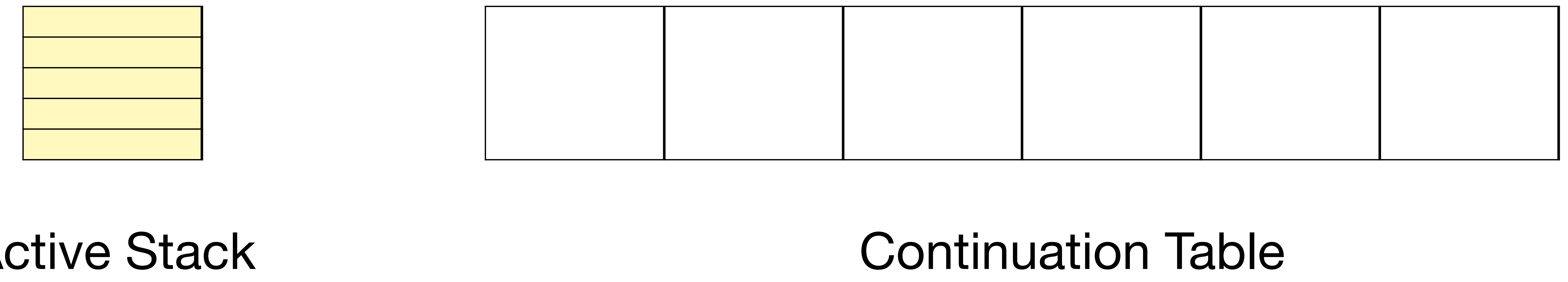
Capture the stack

where s’ = s but with context $L^{\max}$ and v_l^* stored at index k

Execute h in fresh stack. k is a new ID

Current stack is discarded

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ \mathbf{restore}] \rightsquigarrow_i s';\ v_l^{*\prime};\ L^{\max\prime}[(\mathbf{i64.const}\ v)]$$



Wasm/k Semantics

`control` and `restore`

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\mathbf{i64.const}\ v)\ (\mathbf{control}\ h)] \leadsto_i \boxed{s';}\ \boxed{\epsilon;\ (\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ (\mathbf{call}\ h)}\ \mathbf{trap}$$

Capture the stack

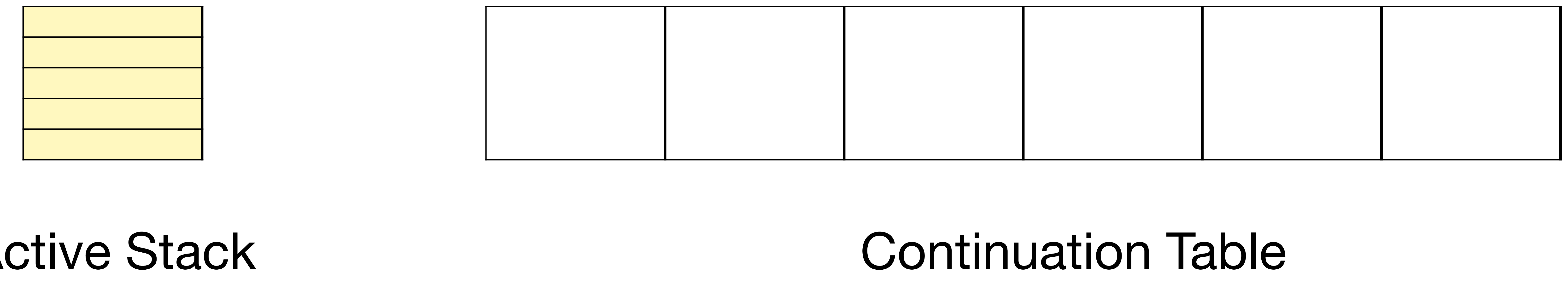
where s’ = s but with context L^{max} and v_l^* stored at index k

Execute h in fresh stack. k is a new ID

Current stack is discarded

Retrieved from s at index k

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ \mathbf{restore}] \leadsto_i s';\ v_l^{*\prime};\ \boxed{L^{\max\prime}}[(\mathbf{i64.const}\ v)]$$

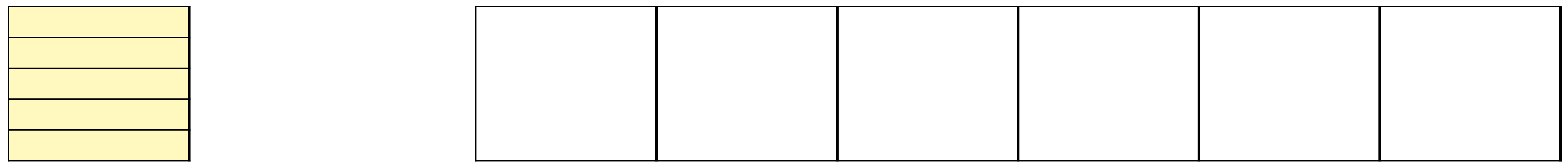


Wasm/k Semantics

`control` and `restore`

$$s;\ v_l^*;\ \underbrace{\boxed{L^{\max}}}_{\text{Capture the stack}}[(\mathbf{i64.const}\ v)\ (\mathbf{control}\ h)] \leadsto_i \underbrace{\boxed{s';}}_{\text{where s' = s but with context } L^{\max} \text{ and } v_l^{*} \text{ stored at index k}}\ \underbrace{\boxed{\epsilon;\ (\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ (\mathbf{call}\ h)}}_{\text{Execute h in fresh stack. k is a new ID}}\ \mathbf{trap}$$

$$s;\ v_l^*;\ \overbrace{\boxed{L^{\max}}}^{\text{Current stack is discarded}}[(\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ \mathbf{restore}] \leadsto_i \overbrace{\boxed{s';}}^{\text{Index k is marked as free}}\ v_l^{*\prime};\ \overbrace{\boxed{L^{\max\prime}}}^{\text{Retrieved from s at index k}}[(\mathbf{i64.const}\ v)]$$



Wasm/k Semantics

`control` and `restore`

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\mathbf{i64.const}\ v)\ (\mathbf{control}\ h)] \leadsto_i \boxed{s';}\ \boxed{\epsilon;\ (\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ (\mathbf{call}\ h)}\ \mathbf{trap}$$

Capture *entire* stack

where s' = s but with context L^{max} and v_l^* stored at index k

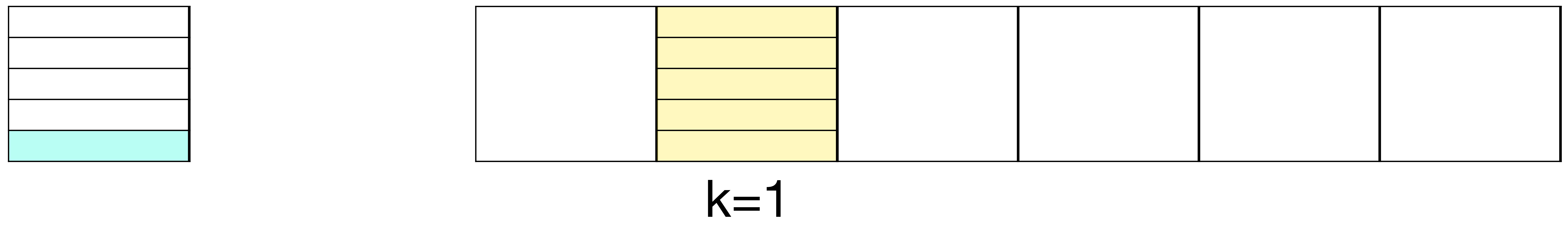
Execute h in fresh stack.
k is a new ID

Current stack is discarded

Index k is marked as free

Retrieved from s at index k

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\mathbf{i64.const}\ \kappa)\ (\mathbf{i64.const}\ v)\ \mathbf{restore}] \leadsto_i \boxed{s';}\ v_l^{*\prime};\ \boxed{L^{\max\prime}}[(\mathbf{i64.const}\ v)]$$



Active Stack

Continuation Table

Wasm/k Semantics

`control` and `restore`

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\textbf{i64.const}\ v)\ (\textbf{control}\ h)] \rightsquigarrow_i \boxed{s';}\ \boxed{\epsilon;\ (\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ (\textbf{call}\ h)}\ \textbf{trap}$$

Capture the stack

where s’ = s but with context L^{max} and v_l^* stored at index k

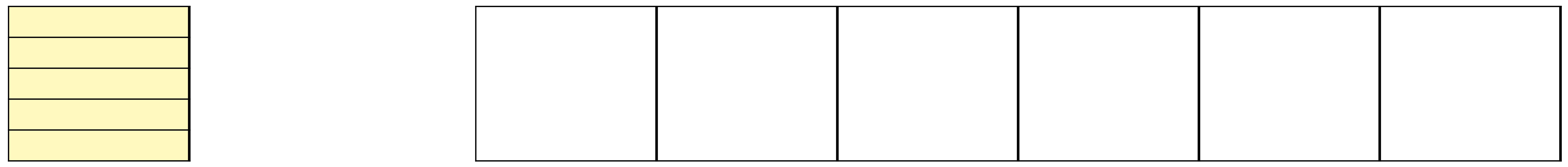
Execute h in fresh stack.
k is a new ID

Current stack is discarded

Index k is marked as free

Retrieved from s at index k

$$s;\ v_l^*;\ \boxed{L^{\max}}[(\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ \textbf{restore}] \rightsquigarrow_i \boxed{s';}\ v_l^{*\prime};\ \boxed{L^{\max\prime}}[(\textbf{i64.const}\ v)]$$



Active Stack

Continuation Table

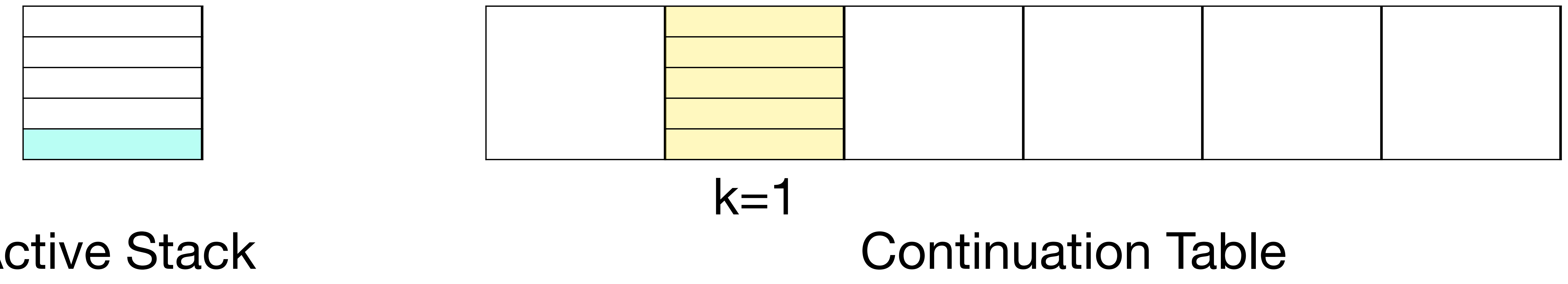
Wasm/k Semantics

`continuation_copy`

$$s;\ \boxed{(\text{i64.const}\ \kappa)}\ \text{continuation\_copy} \hookrightarrow_i \boxed{s';\ (\text{i64.const}\ \kappa')}$$

Stack at index k will be copied

Stack at k is copied to k', and k' is returned



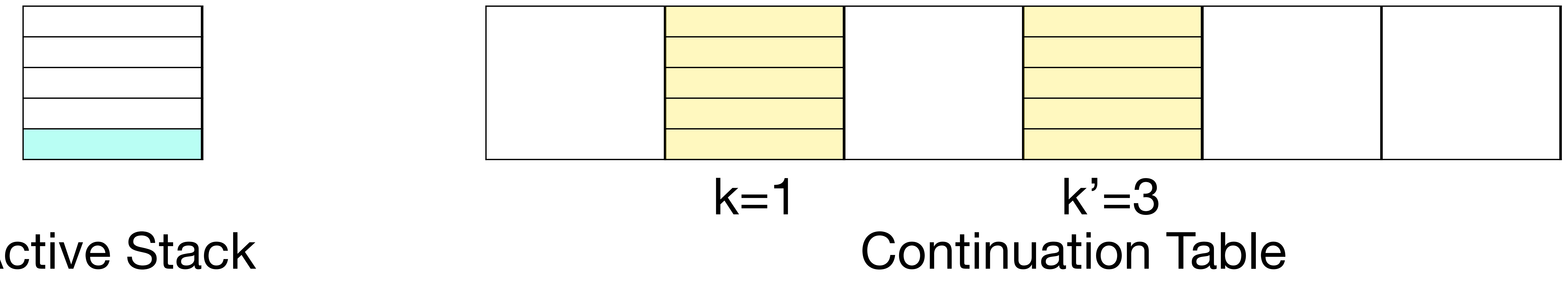
Wasm/k Semantics

`continuation_copy`

$$s;\ \boxed{(\text{i64.const}\ \kappa)}\ \text{continuation\_copy} \hookrightarrow_i \boxed{s';\ (\text{i64.const}\ \kappa')}$$

Stack at index k will be copied

Stack at k is copied to k', and k' is returned

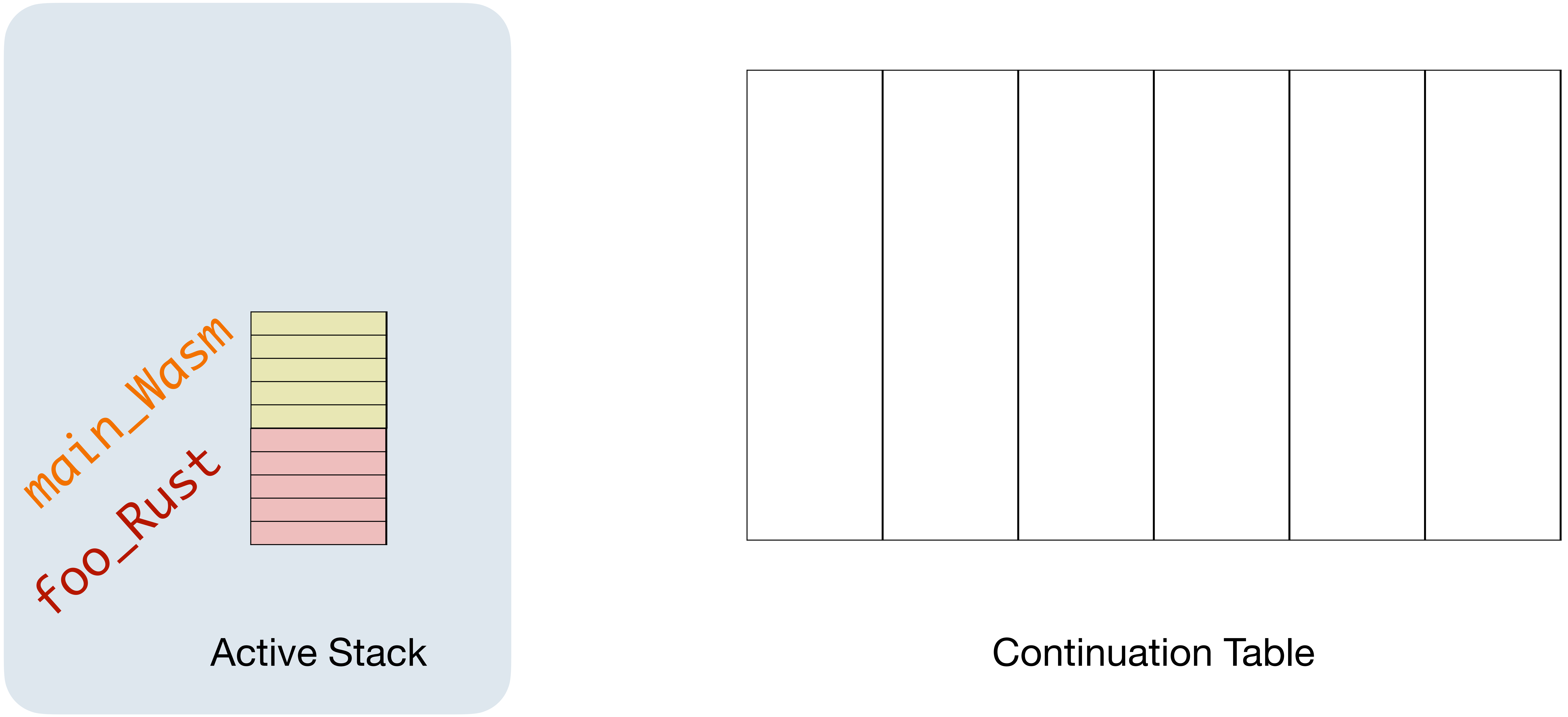


What We Have so Far

C/k = C +		Wasm/k = Wasm +
`control()`	$\longrightarrow$	`control`
`restore()`	$\longrightarrow$	`restore`
`continuation_copy()`	$\longrightarrow$	`continuation_copy`
`continuation_delete()`	$\longrightarrow$	`continuation_delete`
`prompt()`	$\longrightarrow$	`prompt … end`

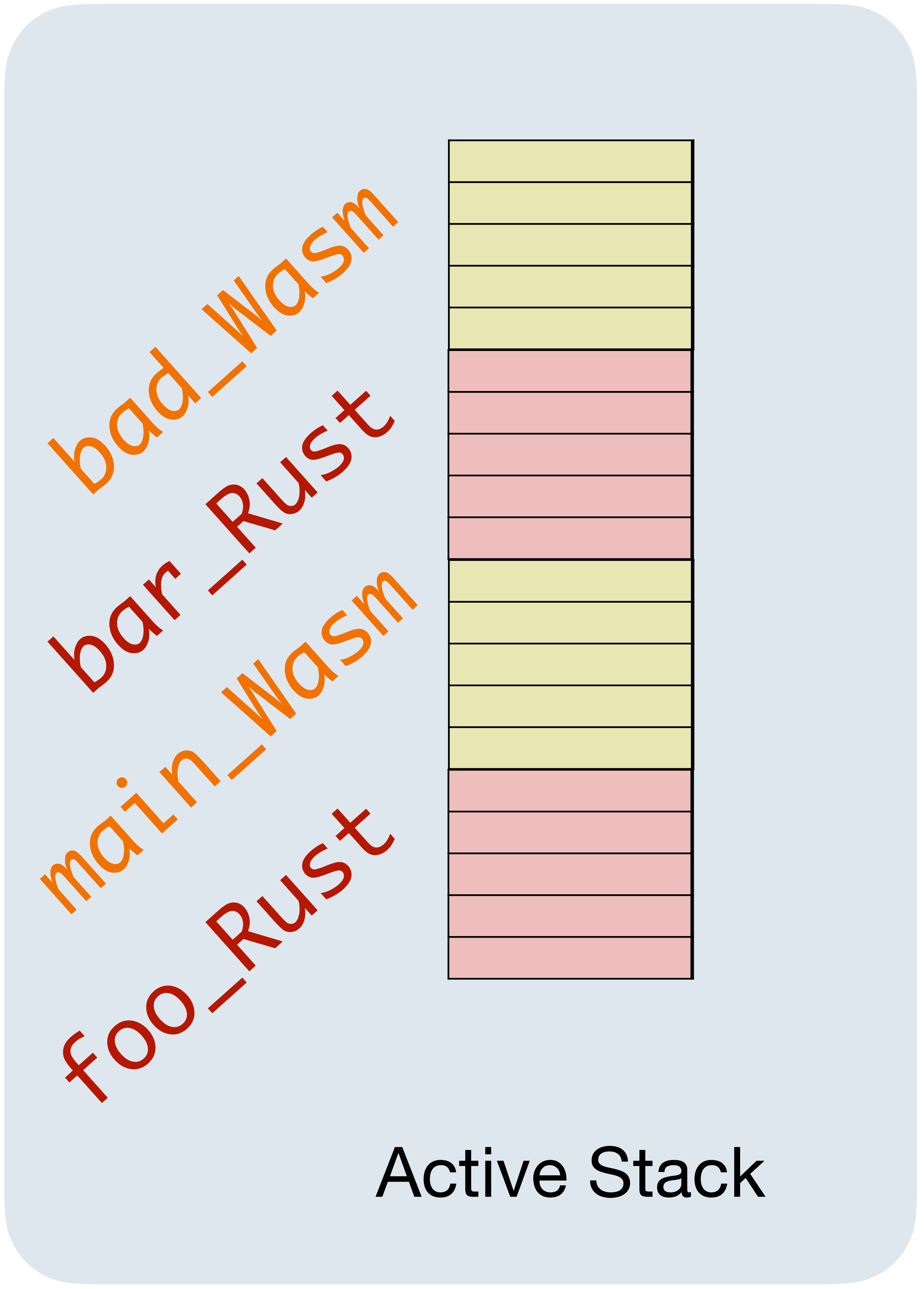
Is it Safe?

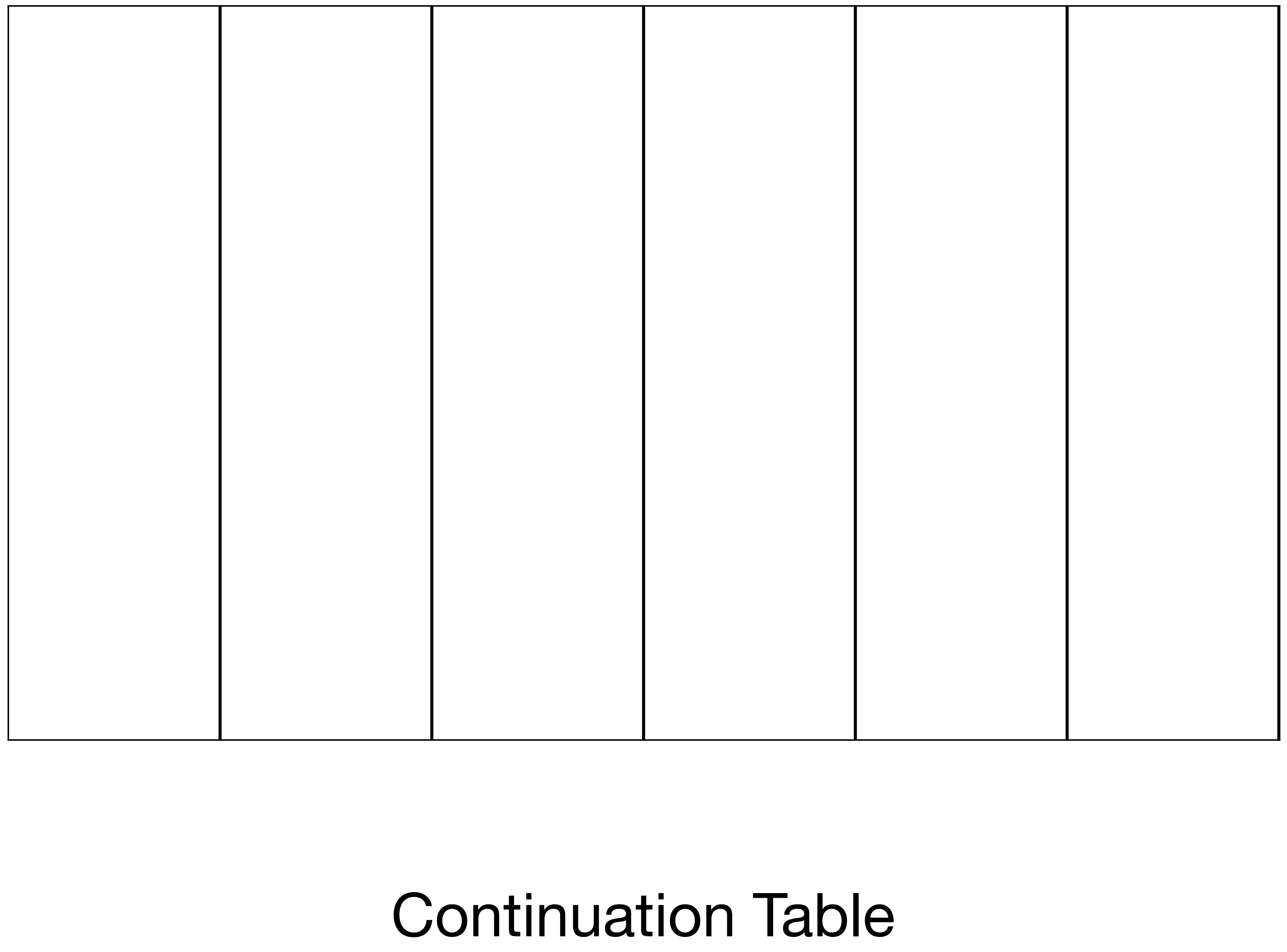
Almost… what about FFI?



Is it Safe?

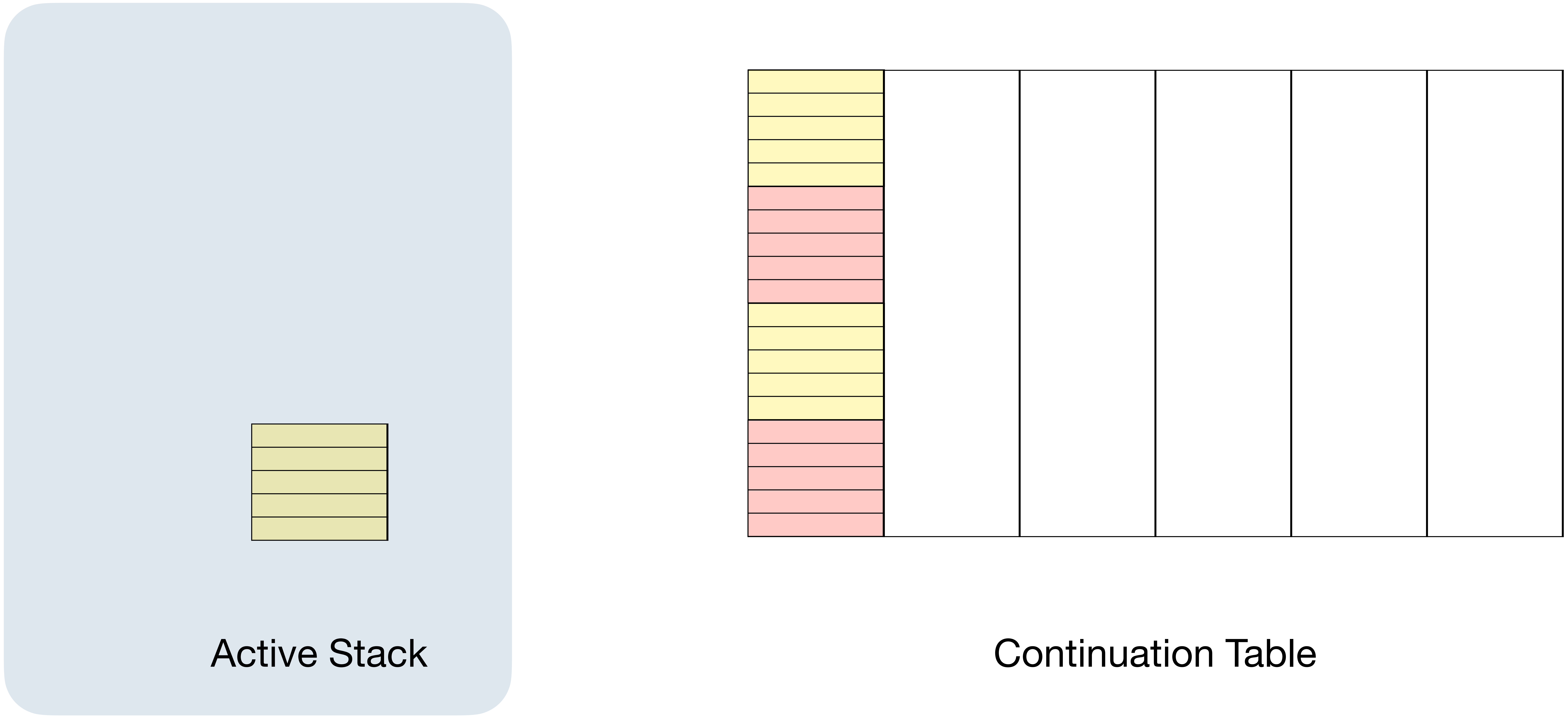
Almost… what about FFI?





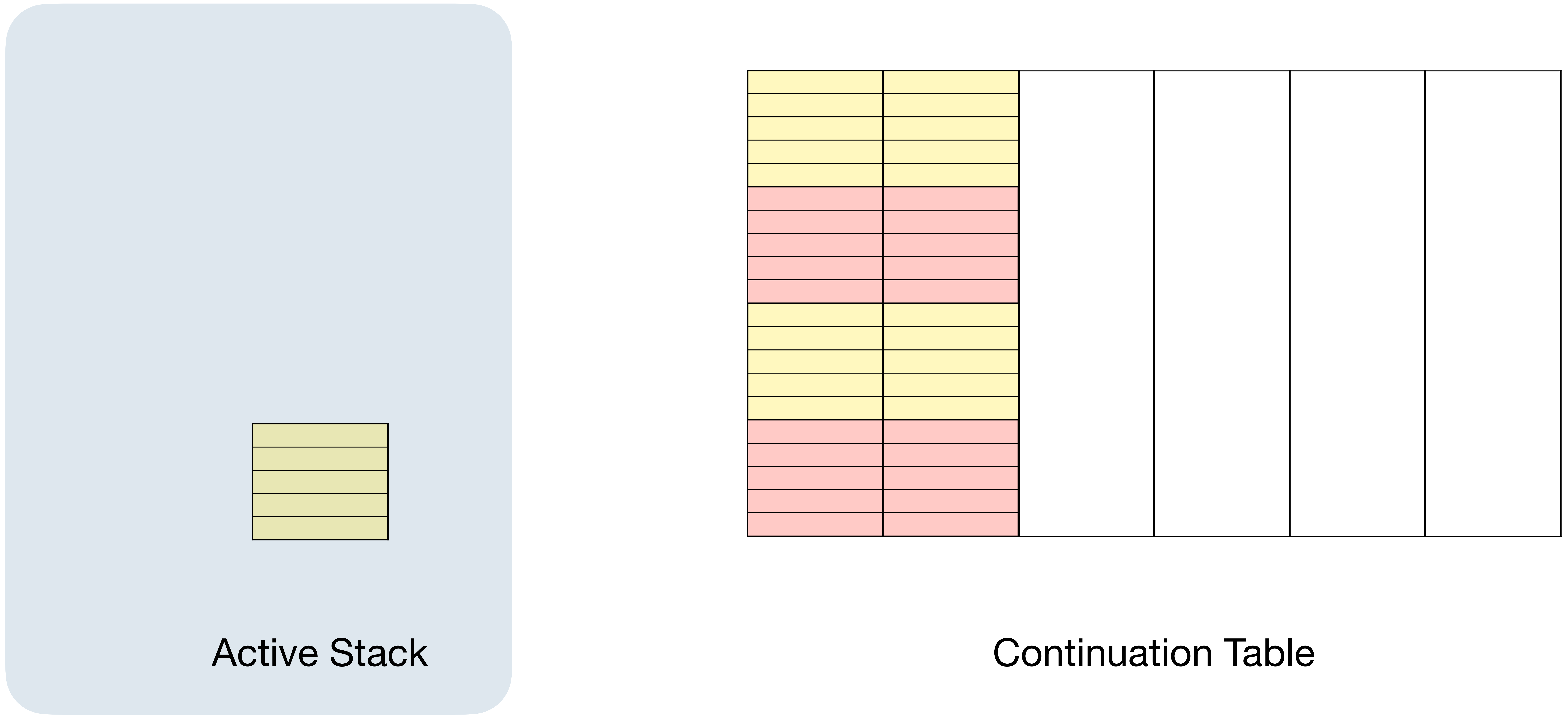
Is it Safe?

Almost… what about FFI?



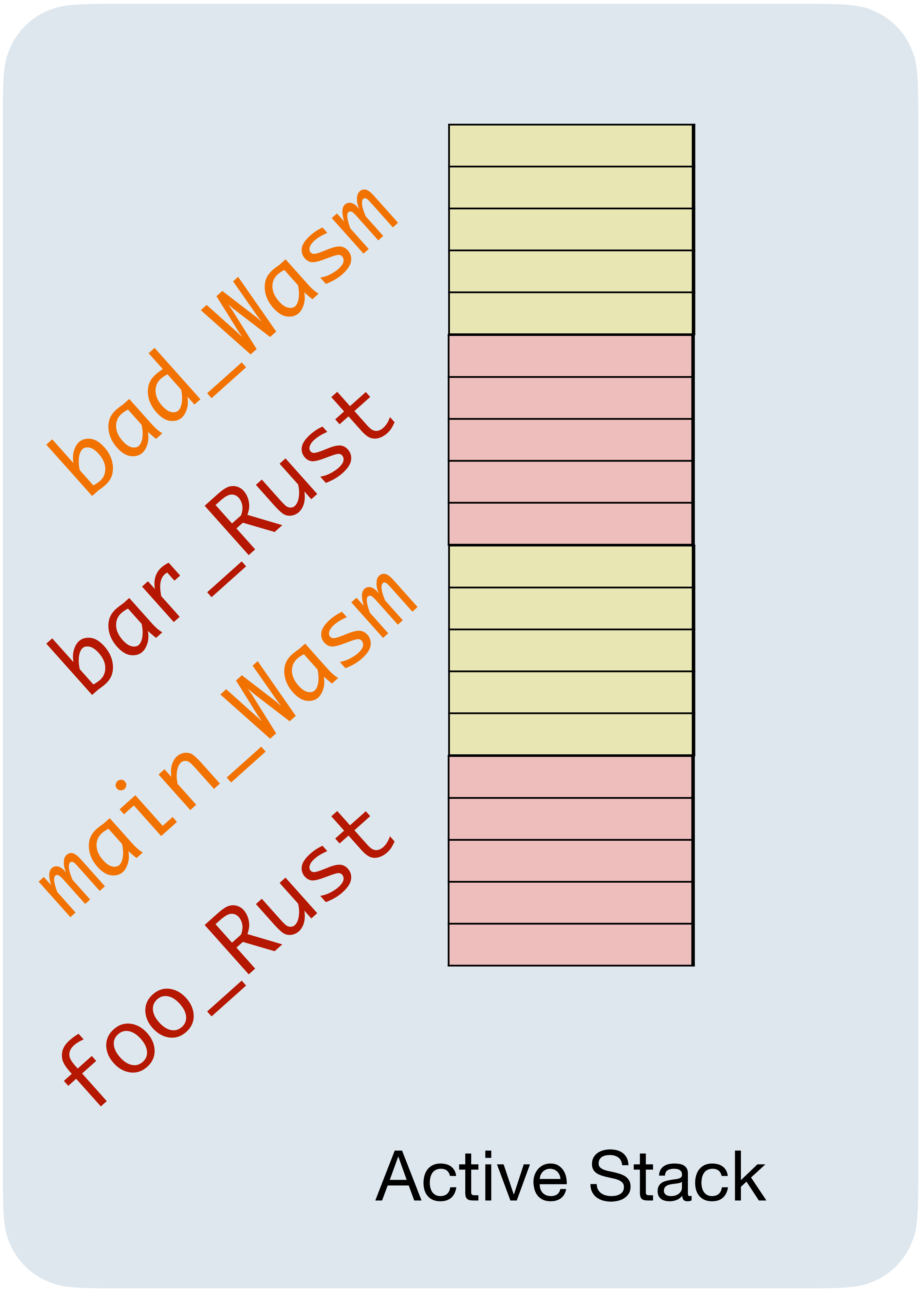
Is it Safe?

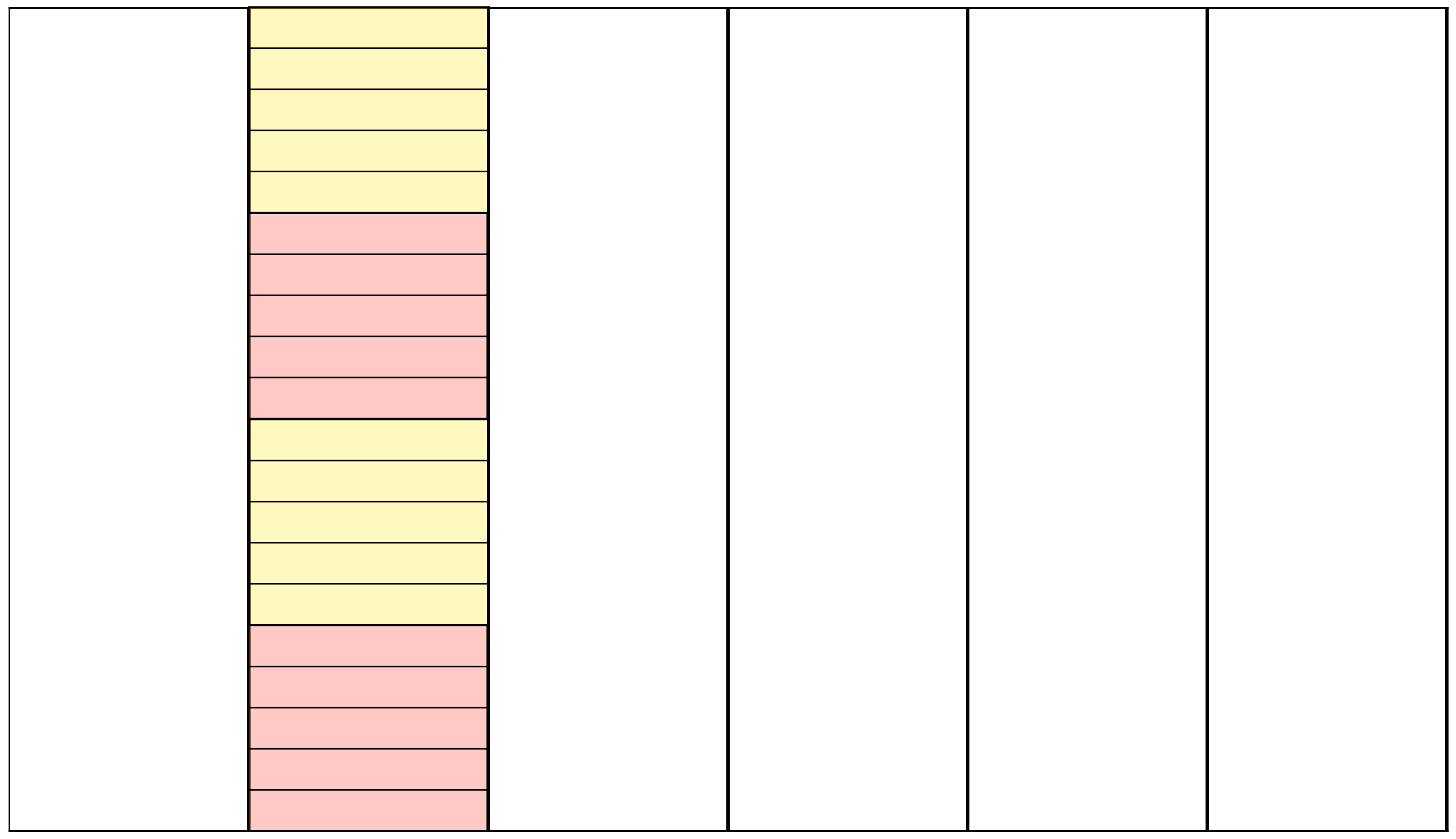
Almost… what about FFI?



Is it Safe?

Almost… what about FFI?

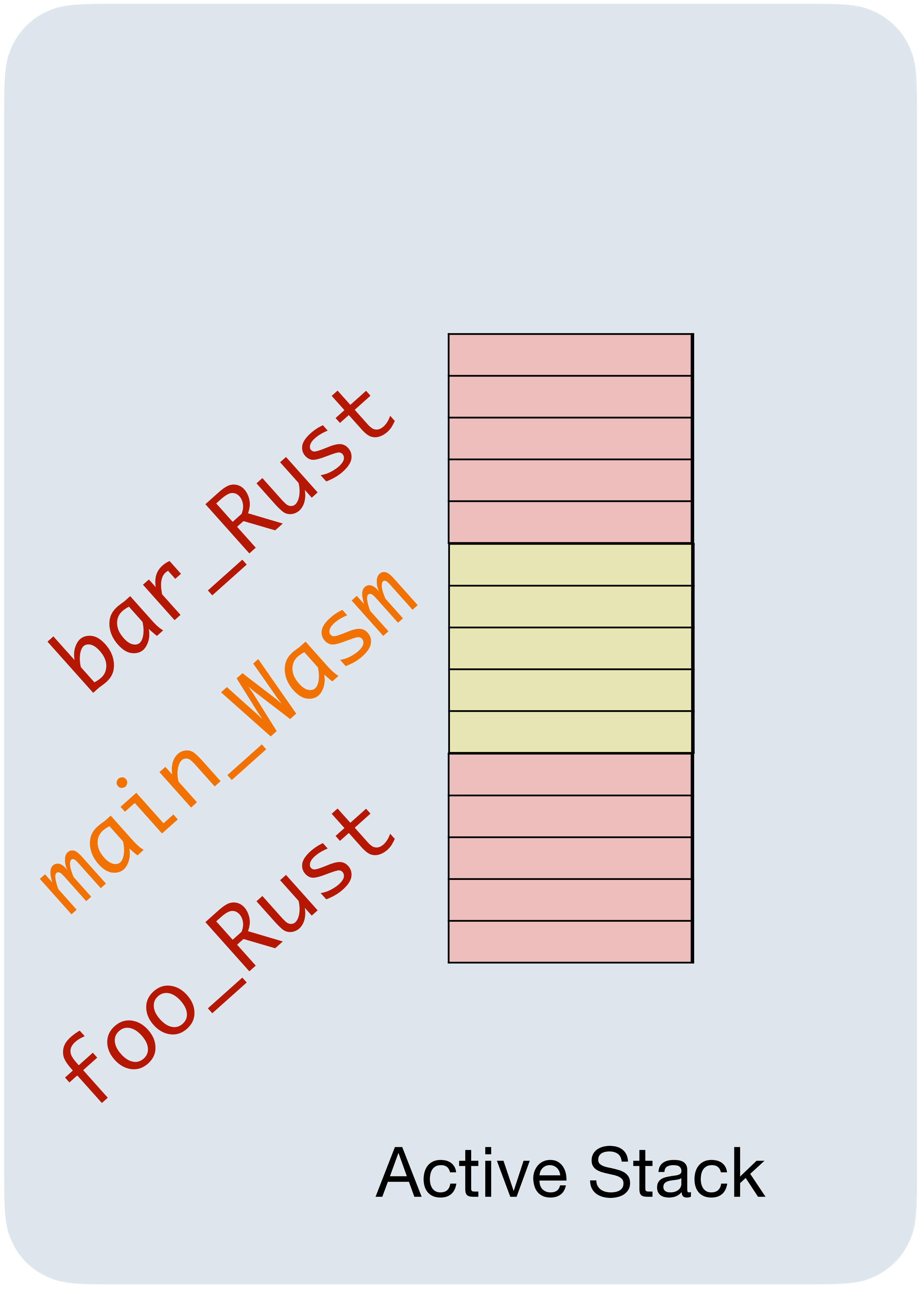


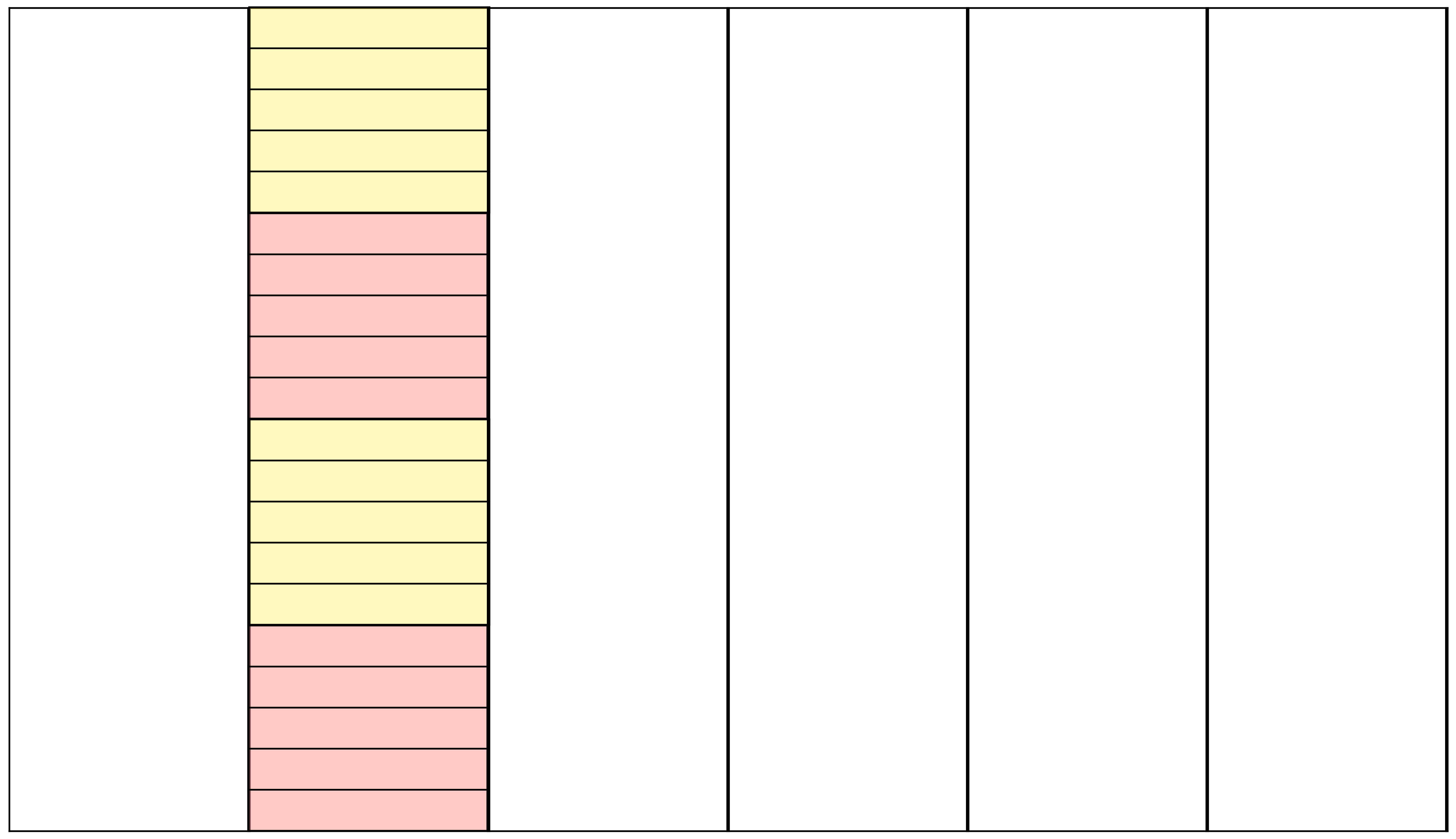


Continuation Table

Is it Safe?

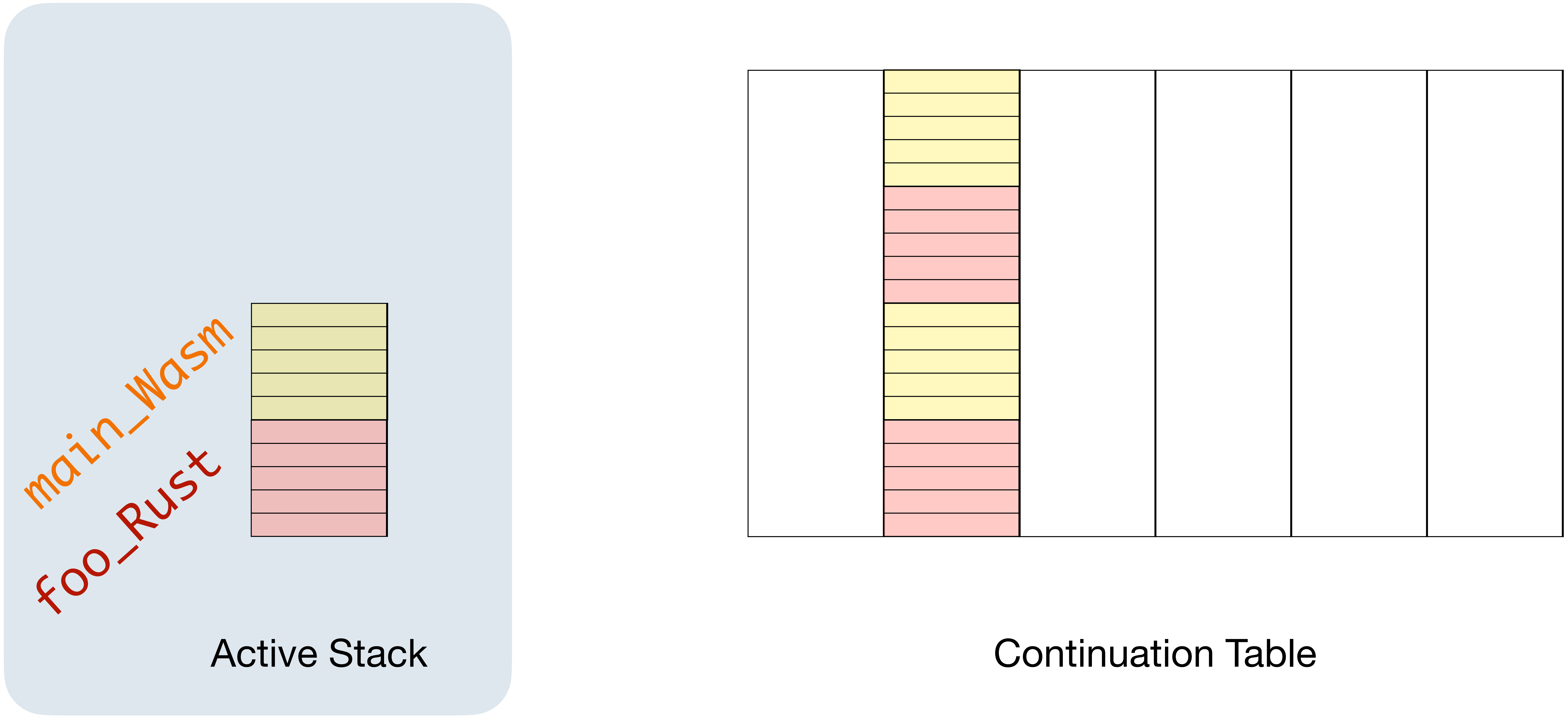
Almost… what about FFI?





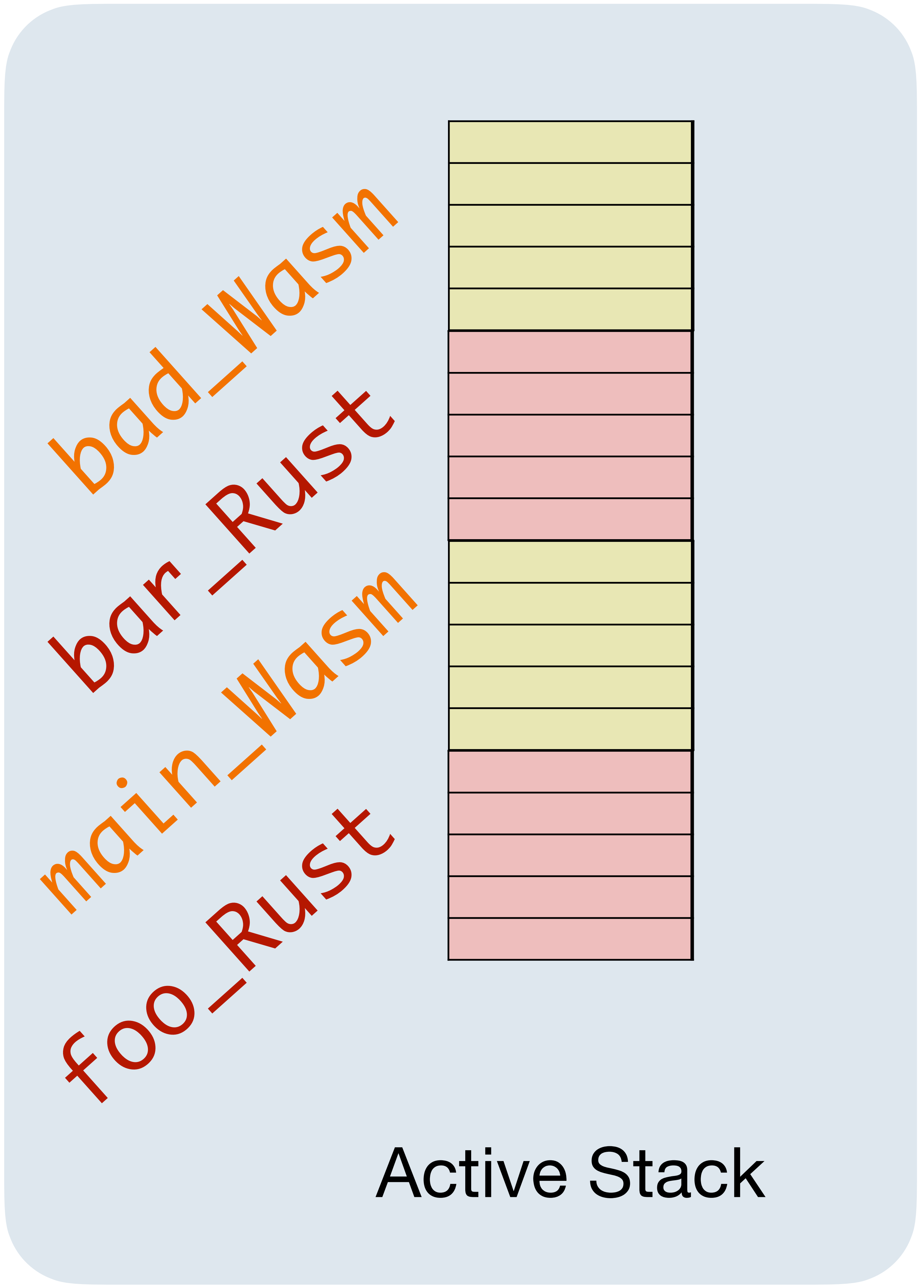
Is it Safe?

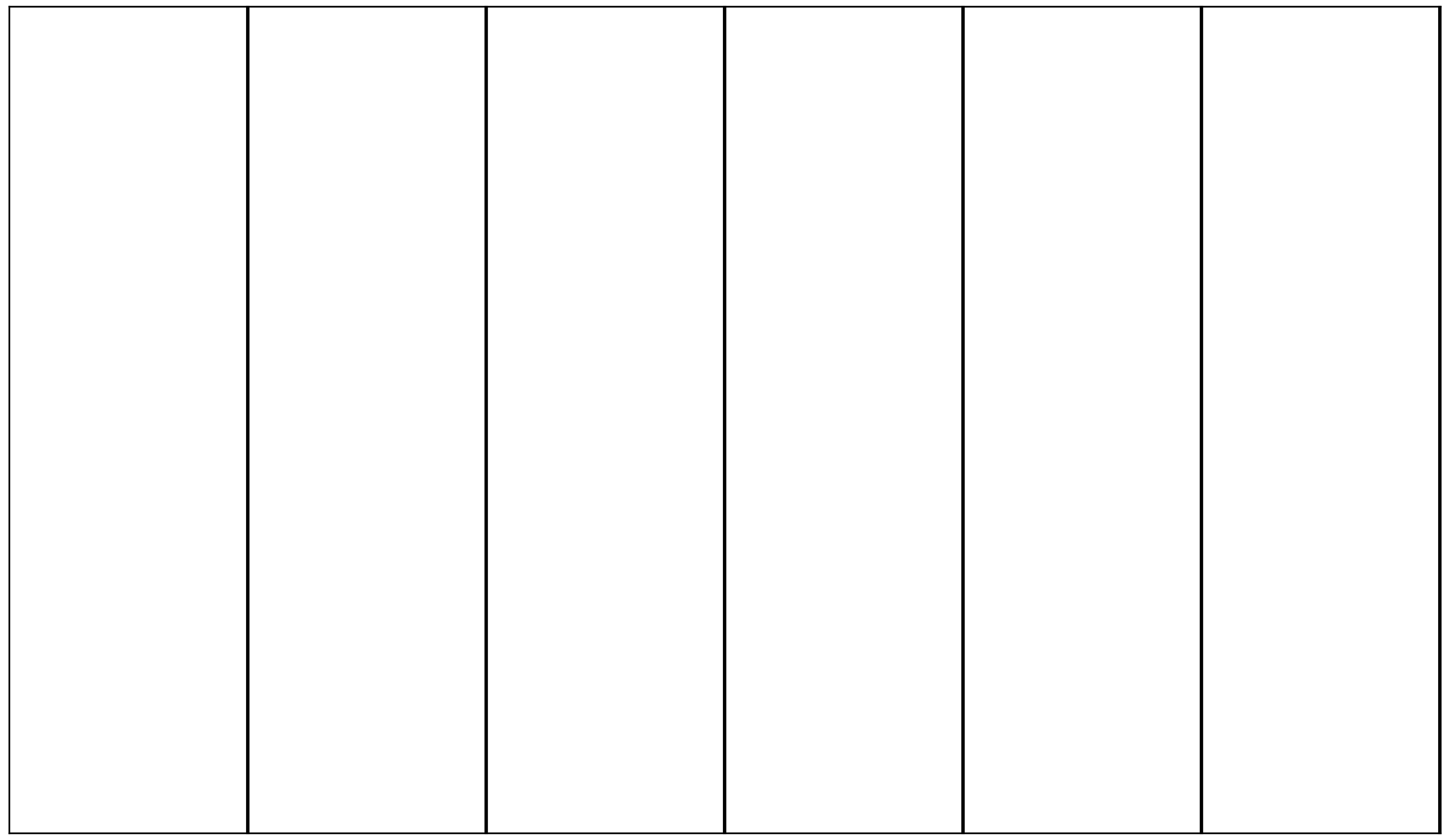
Almost… what about FFI?



Is it Safe?

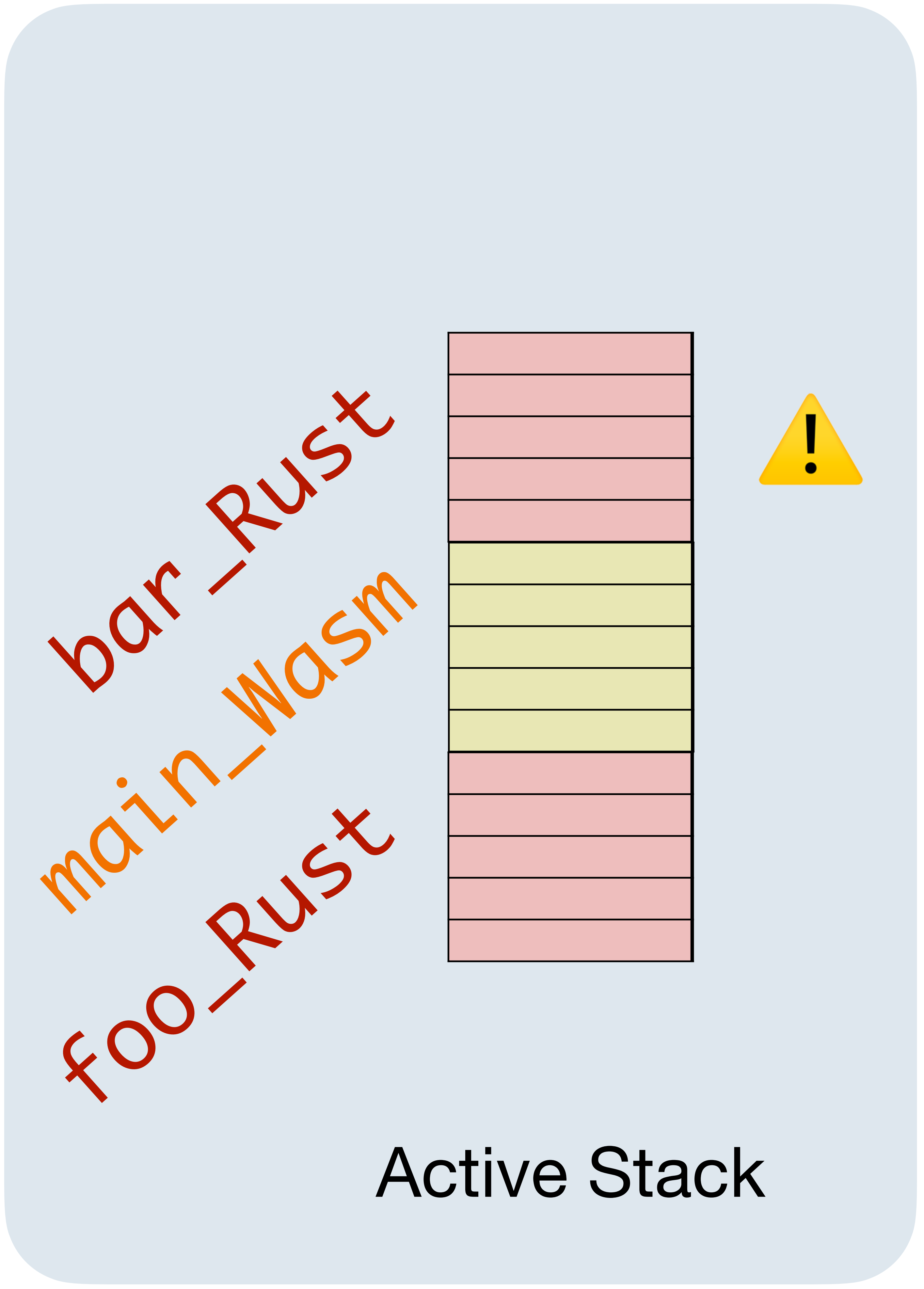
Almost… what about FFI?

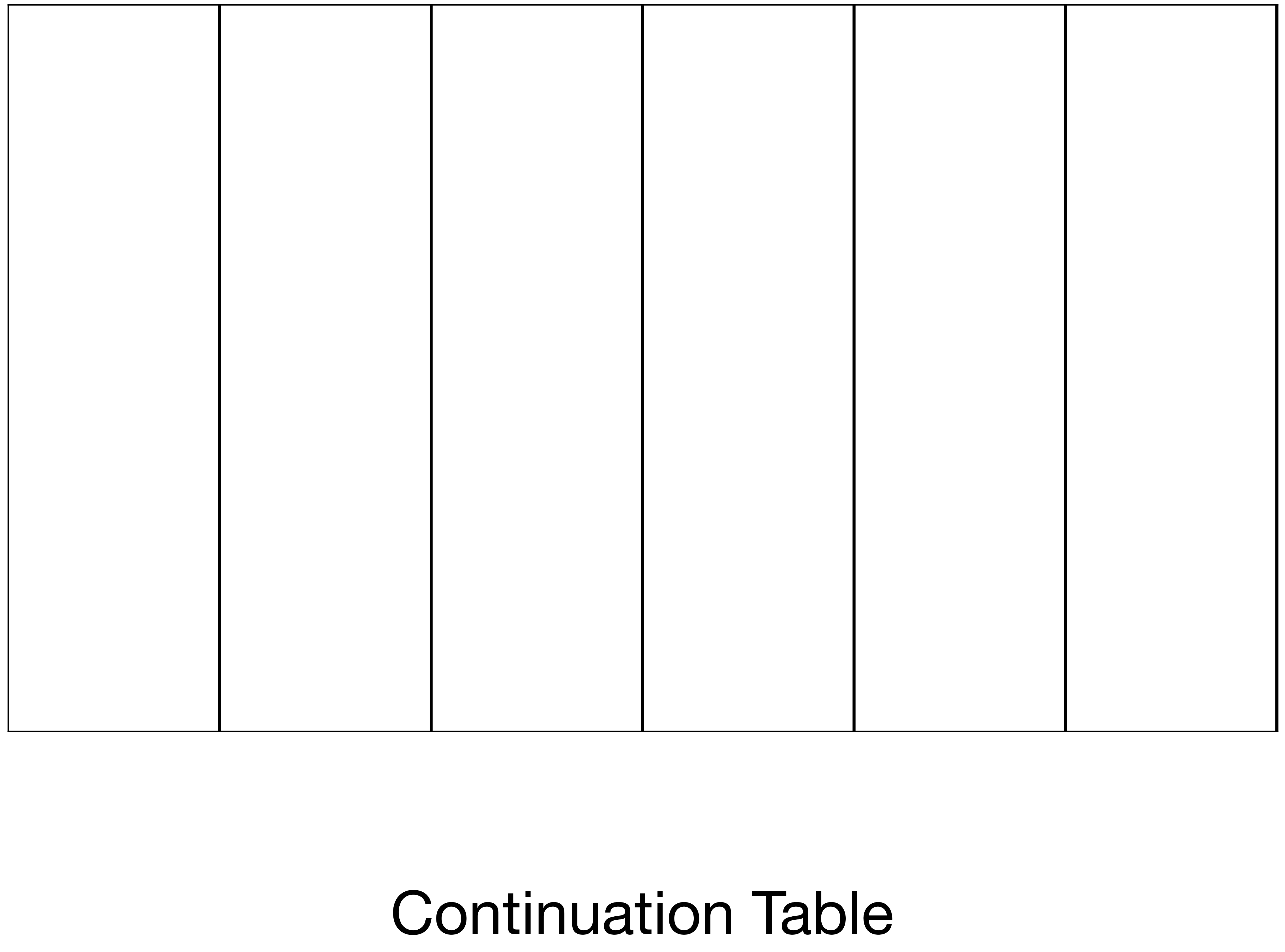




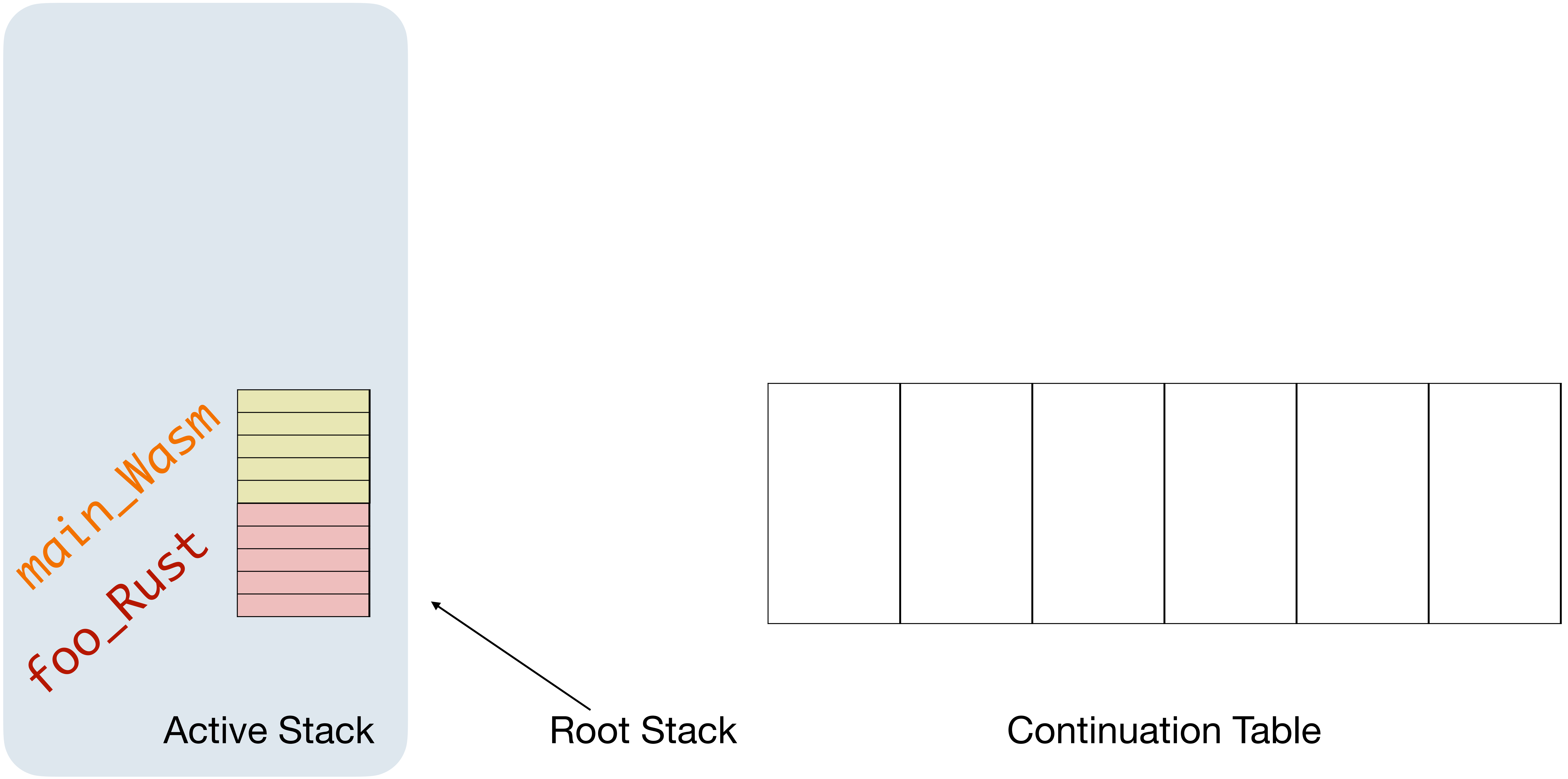
Is it Safe?

Almost… what about FFI?

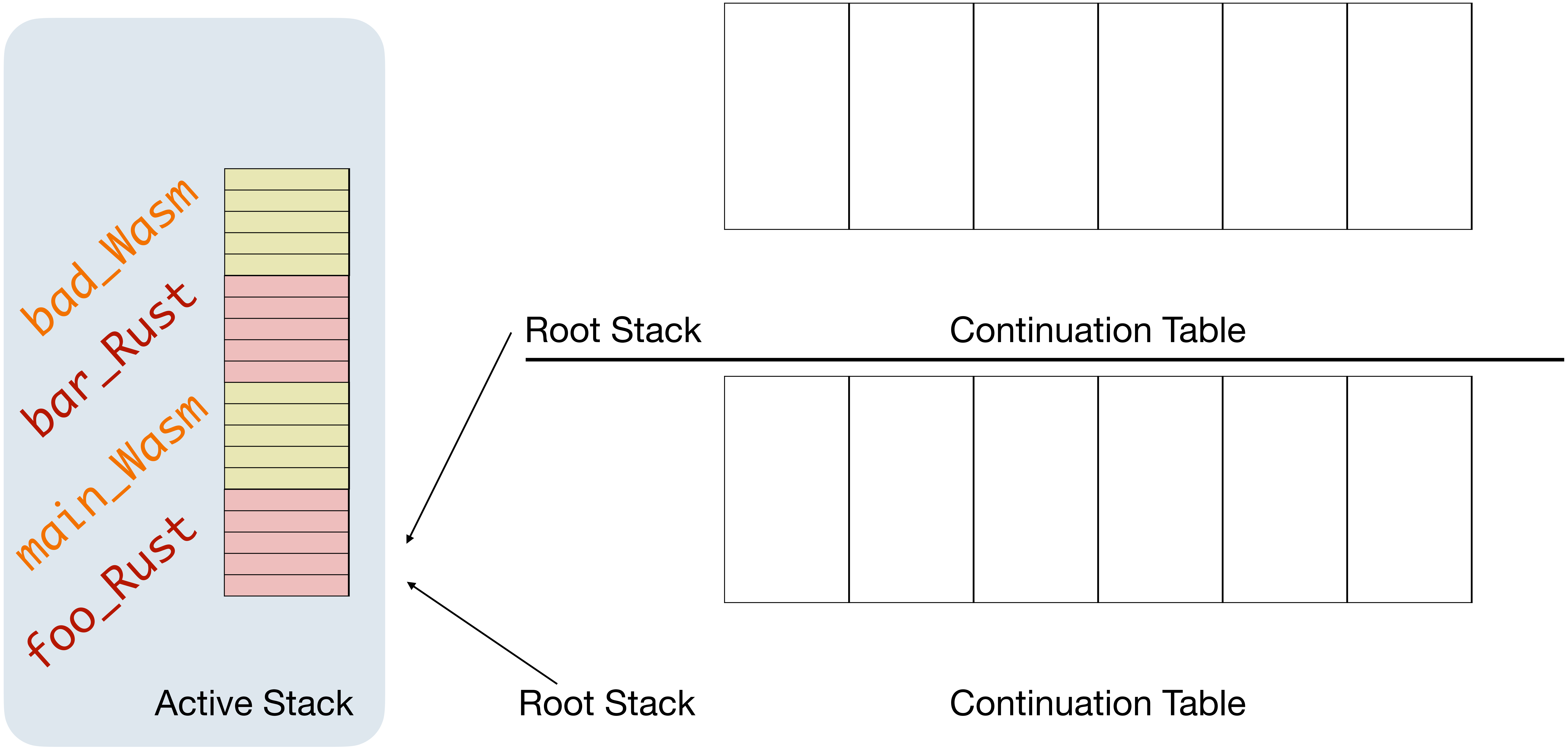




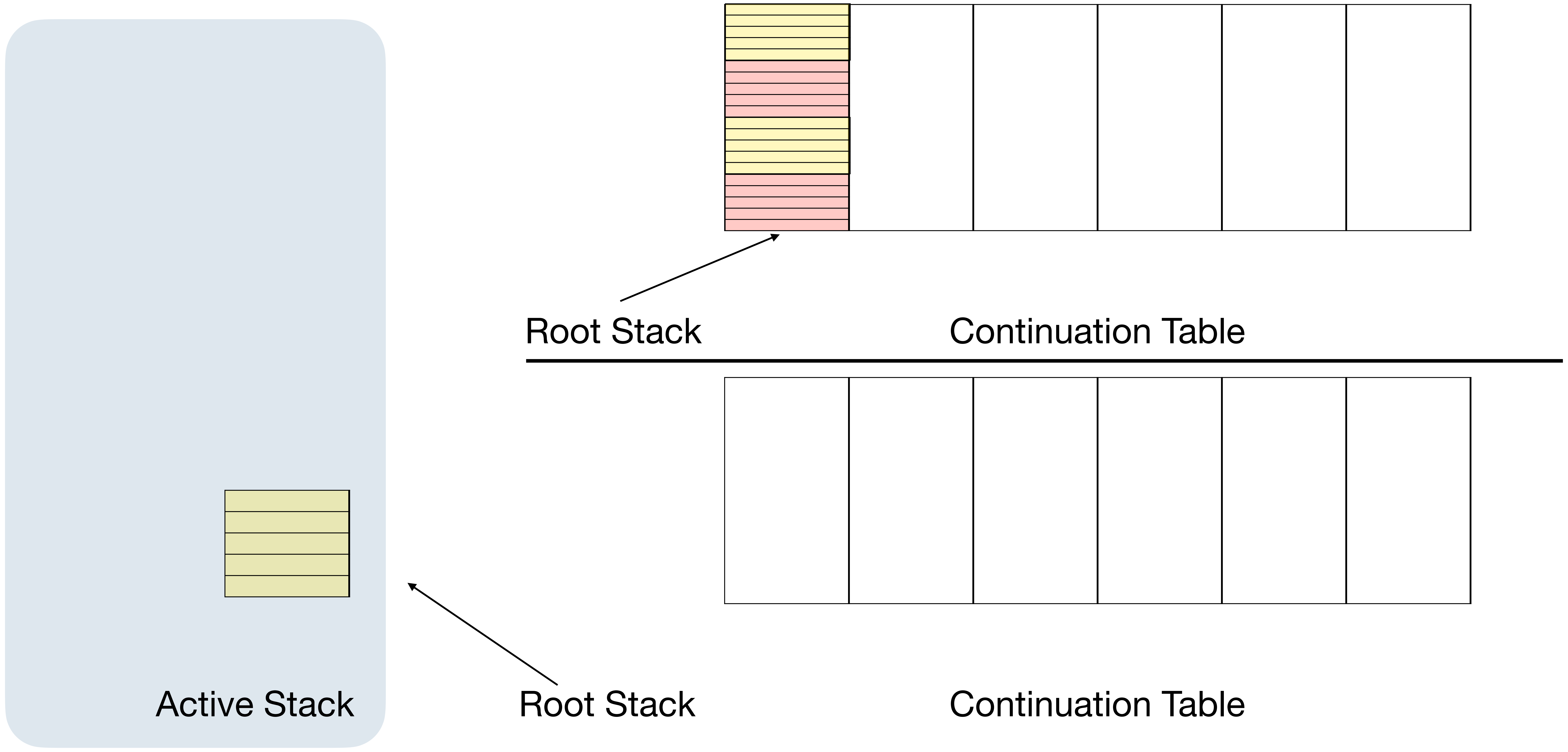
Solution: Delimit at FFI Boundaries



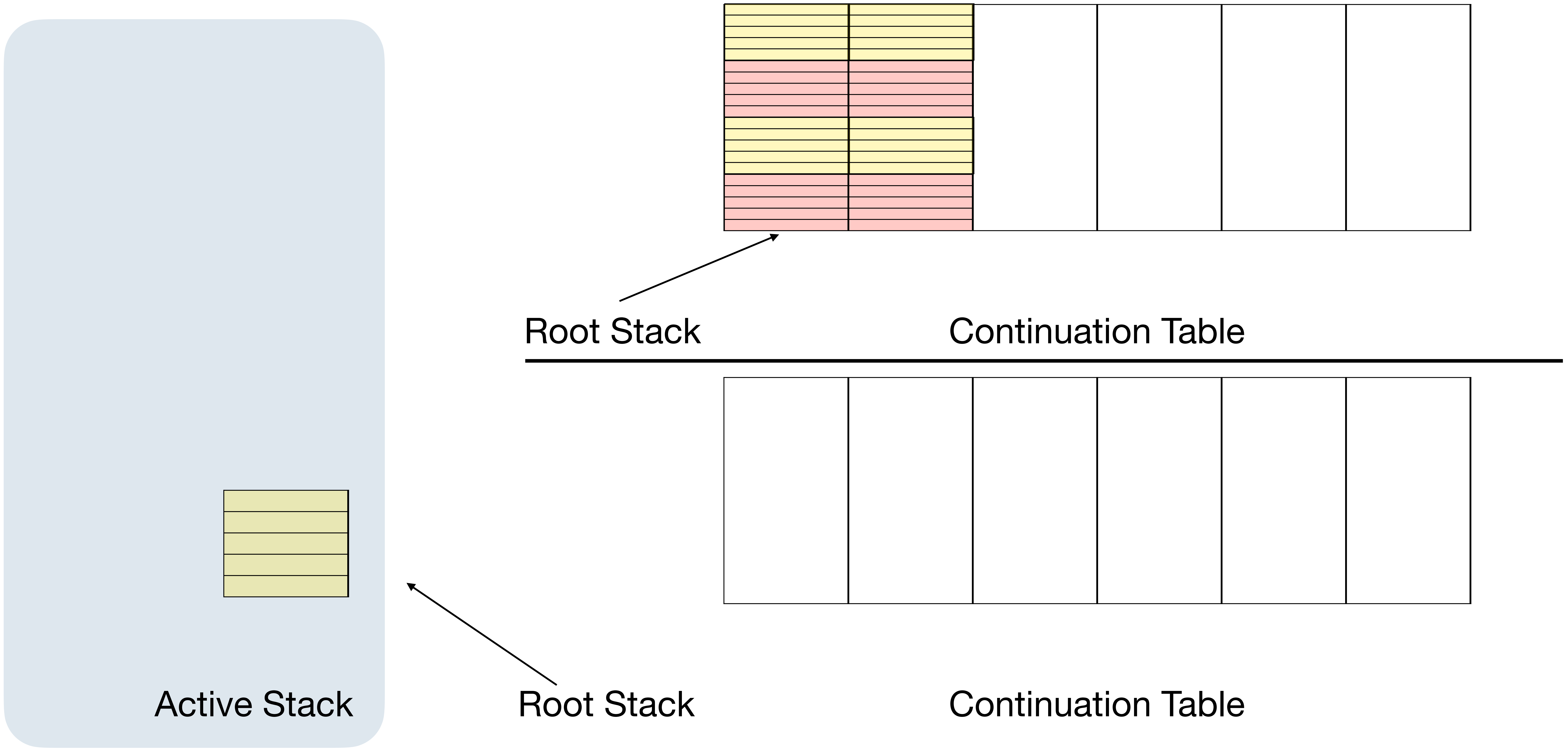
Solution: Delimit at FFI Boundaries



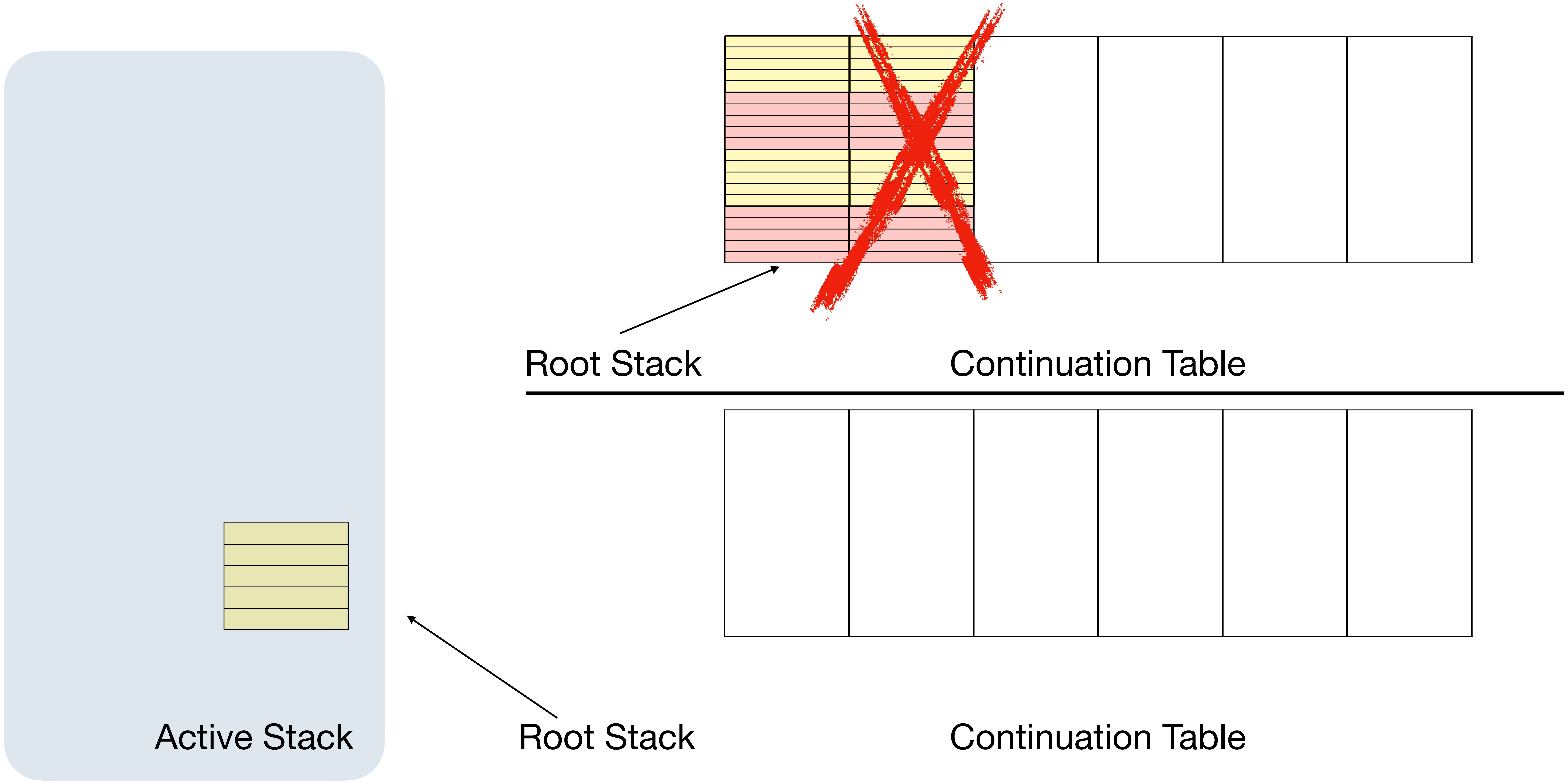
Solution: Delimit at FFI Boundaries



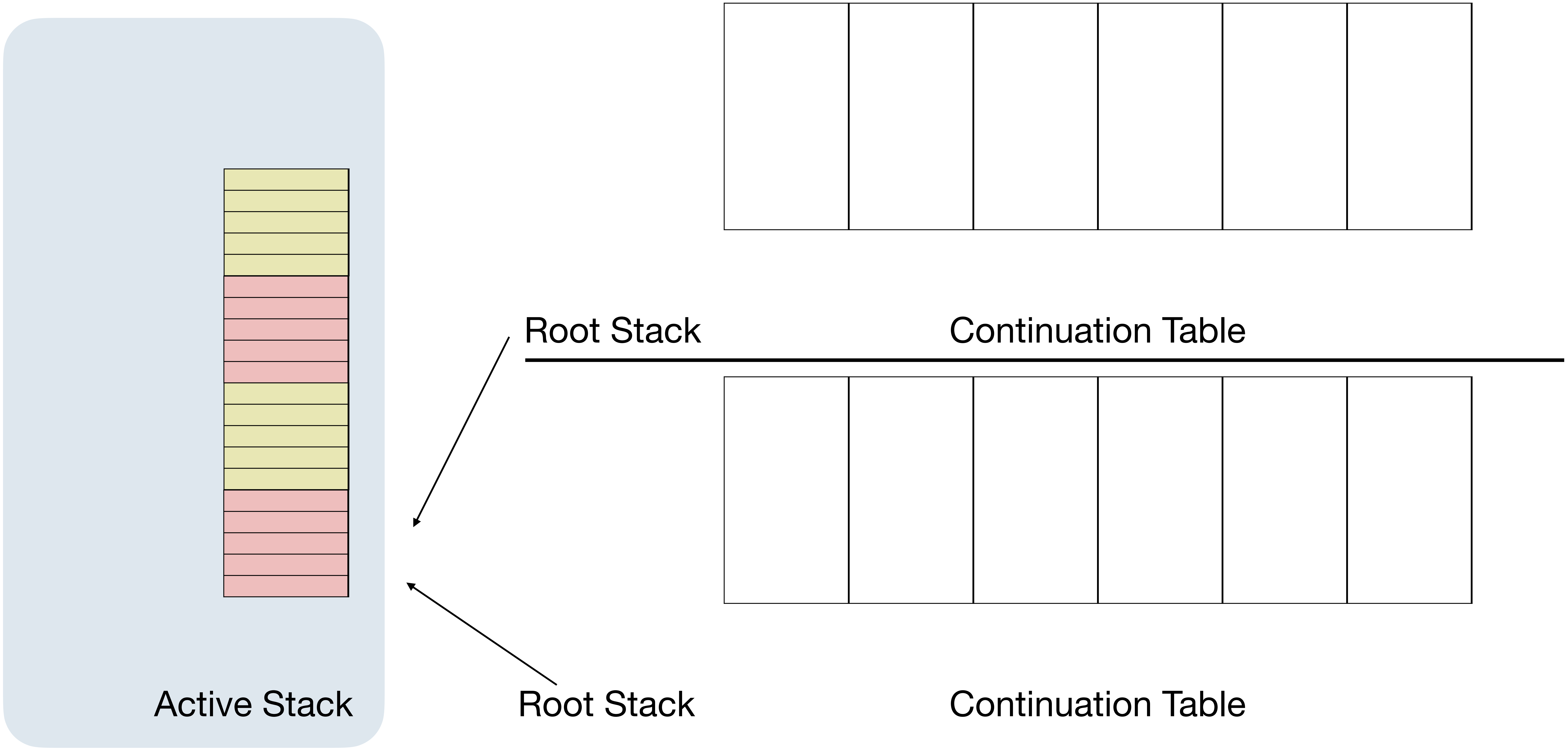
Solution: Delimit at FFI Boundaries



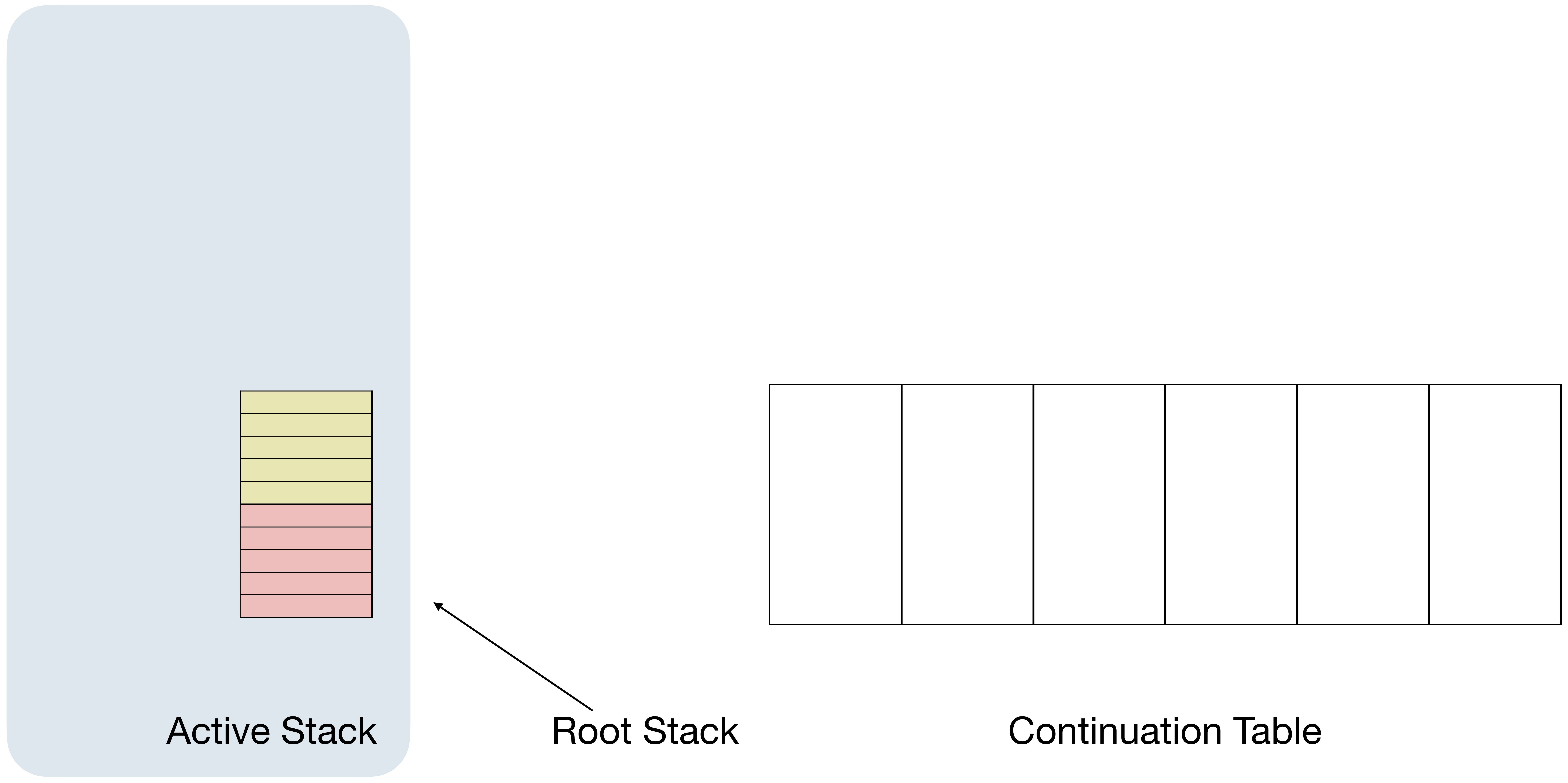
Solution: Delimit at FFI Boundaries



Solution: Delimit at FFI Boundaries



Solution: Delimit at FFI Boundaries



$$\boxed{s;\ v^*;\ e^* \rightsquigarrow_i s;\ v^*;\ e^*}$$

$$[\text{Cong}]\ \frac{s;\ v^*;\ e^* \hookrightarrow_i s';\ v'^*;\ e'^*}{s;\ v^*;\ L^k[e^*] \hookrightarrow_i s';\ v'^*;\ L^k[e'^*]} \qquad [\text{No-Ctrl}]\ \frac{s;\ v^*;\ e^* \hookrightarrow_i s';\ v'^*;\ e'^*}{s;\ v^*;\ e^* \rightsquigarrow_i s';\ v'^*;\ e'^*}$$

[Ctrl]	$s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ v)\ (\textbf{control}\ h)] \rightsquigarrow_i s';\ \epsilon;\ (\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ (\textbf{call}\ h)\ \textbf{trap}$	if $(s', \kappa) = \delta_{\text{ctrl}}(s, i, v_l^*, L^{\max})$
[Restore]	$s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ \textbf{restore}] \rightsquigarrow_i s';\ v_l^{*\prime};\ L^{\max\prime}[(\textbf{i64.const}\ v)]$	if $(s', v_l^*, L^{\max\prime}) = \delta_{\text{rest}}(s, i, \kappa)$
[Restore-Err]	$s;\ v_l^*;\ L^{\max}[(\textbf{i64.const}\ \kappa)\ (\textbf{i64.const}\ v)\ \textbf{restore}] \rightsquigarrow_i s;\ v_l^*;\ \textbf{trap}$	otherwise
[Copy]	$s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation\_copy} \hookrightarrow_i s';\ (\textbf{i64.const}\ \kappa')$	if $(s', \kappa') = \delta_{\text{copy}}(s, i, \kappa)$
[Copy-Err]	$s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation\_copy} \hookrightarrow_i s;\ \textbf{trap}$	otherwise
[Delete]	$s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation\_delete} \hookrightarrow_i s';\ \epsilon$	if $s' = \delta_{\text{delete}}(s, i, \kappa)$
[Delete-Err]	$s;\ (\textbf{i64.const}\ \kappa)\ \textbf{continuation\_delete} \hookrightarrow_i s;\ \textbf{trap}$	otherwise
[Prompt]	$s;\ \textbf{prompt}\ tf\ e^*\ \textbf{end} \hookrightarrow_i s';\ \textbf{block}\ tf\ e^*\ \textbf{end}\ \textbf{prompt\_end}$	if $s' = \delta_{\text{p}}(s, i)$
[Prompt-End]	$s;\ \textbf{prompt\_end} \hookrightarrow_i s';\ \epsilon$	if $s' = \delta_{\text{p-end}}(s, i)$

$$\delta_{\text{ctrl}}(s, i, v_l^*, L^{\max}) ::= \begin{cases} (\text{setCont}(\text{setRoot}(s, i, \kappa), i, \kappa, \{\text{locals} = v_l^*, \text{ctx} = L^{\max}, \text{inst} = i\}),\ \kappa) & \text{if getRoot}(s, i) = nil \\ (\text{setCont}(s, i, \kappa, \{\text{locals} = v_l^*, \text{ctx} = L^{\max}, \text{inst} = i\}),\ \kappa) & \text{if getRoot}(s, i) \neq nil \end{cases}$$

where κ is fresh, i.e., $\text{getCont}(s, i, \kappa) = nil$

$$\delta_{\text{rest}}(s, i, \kappa) ::= \begin{cases} (\text{setRoot}(\text{setCont}(s, i, \kappa, nil), i, nil),\ \text{getCont}(s, i, \kappa)_{\text{locals}},\ \text{getCont}(s, i, \kappa)_{\text{ctx}}) & \text{if getRoot}(s, i) = \kappa \\ (\text{setCont}(s, i, \kappa, nil),\ \text{getCont}(s, i, \kappa)_{\text{locals}},\ \text{getCont}(s, i, \kappa)_{\text{ctx}}) & \text{if } nil \neq \text{getRoot}(s, i) \neq \kappa \end{cases}$$

$$\delta_{\text{copy}}(s, i, \kappa) ::= (\text{setCont}(s, i, \kappa', \text{getCont}(s, i, \kappa)),\ \kappa') \qquad \text{if getRoot}(s, i) \neq \kappa \wedge \text{getCont}(s, i, \kappa) \neq nil$$

where κ' is fresh, i.e., $\text{getCont}(s, i, \kappa') = nil$

$$\delta_{\text{delete}}(s, i, \kappa) ::= \text{setCont}(s, i, \kappa, nil) \qquad \text{if getRoot}(s, i) \neq \kappa \wedge \text{getCont}(s, i, \kappa) \neq nil$$

$$\delta_{\text{p}}(s, i) ::= s' \text{ where } s' = s \text{ except } s'_{\text{inst}}(i)_{\text{pstack}} \mapsto \text{push}(s_{\text{inst}}(i)_{\text{pstack}}, \{\text{ctable} = nil^*, \text{root} = nil, \text{inst} = i\})$$

$$\delta_{\text{p-end}}(s, i) ::= s' \text{ where } s' = s \text{ except } s'_{\text{inst}}(i)_{\text{pstack}} \mapsto \text{pop}(s_{\text{inst}}(i)_{\text{pstack}}) \qquad \text{if getRoot}(s, i) = nil$$

$$\text{getRoot}(s, i) ::= \text{top}(s_{\text{inst}}(i)_{\text{pstack}})_{\text{root}}$$

$$\text{getCont}(s, i, \kappa) ::= \text{top}(s_{\text{inst}}(i)_{\text{pstack}})_{\text{ctable}}(\kappa)$$

$$\text{setRoot}(s, i, \kappa_R^?) ::= s' \text{ where } s' = s \text{ except top}(s'_{\text{inst}}(i)_{\text{pstack}})_{\text{root}} \mapsto \kappa_R^?$$

$$\text{setCont}(s, i, \kappa, \gamma^?) ::= s' \text{ where } s' = s \text{ except top}(s'_{\text{inst}}(i)_{\text{pstack}})_{\text{ctable}}(\kappa) \mapsto \gamma^?$$

$$C ::= \{\ldots, \text{label}\ ((t^*)^*)^*, \text{pstack}\{\text{ctable}(t^* \mid nil)^*, \text{root}(\kappa_R \mid nil)\}^*\}$$

$$\frac{C_{\text{func}}(h) = \text{i64 i64} \to \epsilon}{C \vdash (\textbf{control}\ h) : \text{i64} \to \text{i64}}$$

$$\frac{}{C \vdash \textbf{restore} : t_1^*\ \text{i64 i64} \to t_2^*}$$

$$\frac{}{C \vdash \textbf{continuation\_copy} : \text{i64} \to \text{i64}}$$

$$\frac{}{C \vdash \textbf{continuation\_delete} : \text{i64} \to \epsilon}$$

$$\frac{tf = t_1^n \to t_2^m \qquad C\{\text{label} = C_{\text{label}}; ((t_2^m)),\ \text{return} = \epsilon\} \vdash e^* : tf}{C \vdash \textbf{prompt}\ tf\ e^*\ \textbf{end} : tf}$$

Implementation

- How to compile C/k → Wasm/k?
  - Mostly easy: Can be done locally… using C macros + regex find / replace on Emscripten output
  - But, need to also capture / restore the C shadow stack in linear memory
- Implementation of Wasm/k
  - Implemented in a fork of Wasmtime, targeting x86
  - Each instruction (e.g. `control`) calls handwritten x86 assembly to save registers and the stack. Similar to `setjmp` / `longjmp`
  - One-shot continuations means only `continuation_copy` needs to perform a `memcpy`

Thank you!

Full proofs and implementation at:

https://wasmk.github.io

Donald Pinckney (pinckney.d@northeastern.edu), Arjun Guha, Yuriy Brun