“zero alloc” Static Analysis

Greta Yorsh

Tools and Compilers Group

Jane Street

Who or What is Jane Street?

- Proprietary Trading Company
- Trading on financial markets (such as stock exchanges)
- Market Maker / Liquidity Provider
- Using the company's own capital
- 3000+ people in New York, London, Hong Kong, and others
- We scale with technology in every part of our business
- We use OCaml in (almost) every part of our technology

Why OCaml?

- What matters for Jane Street?
- Where does OCaml fit in?
- How does it work in practice?

based on slides by Ron Minsky and Pedro Flemming

Trading is scary

What matters at Jane Street?

- ~~“Move fast and break things”~~
- Correctness
- Performance
- Reliability
- Predictability
- Agility

Where does OCaml fit in?

	Imperative	**Functional**
Dynamic	Python Perl Ruby JavaScript PHP	Lisp Scheme Racket Clojure
Static	C C# Java C++ Fortran	**OCaml** Rust Scala Haskell F# SML

Where does OCaml fit in?

	Imperative	**Functional**
Dynamic	Python Perl Ruby JavaScript PHP	Lisp Scheme Racket Clojure
Static	C C# Java C++ Fortran	**OCaml** Rust Scala Haskell F# SML

OCaml: expressive and lightweight types

- Static types
- Type inference
- Module system
- Algebraic Data Types (ADTs)

How does it work in practice?

- Range of use cases
- Tools and libraries
- What about speed?

Why do we care about allocation?

Why do we care about allocation?

- Most code we write at Jane Street allocates lots
- The compiler is good at optimizing allocations
- Allocation in OCaml is actually pretty fast!

Why do we care about allocation?

- Most code we write at Jane Street allocates lots
- The compiler is good at optimizing allocations
- Allocation in OCaml is actually pretty fast!

Why do we care about allocation?

- Most code we write at Jane Street allocates lots
- The compiler is good at optimizing allocations
- Allocation in OCaml is actually pretty fast!

- Any time you allocate, the GC may run
- GC briefly pauses the application
- Bad for low latency systems

“zero alloc" annotations

“zero alloc" annotations

- Users specify functions that must be “zero alloc”
- Compiler conservatively checks the annotations

“zero alloc" annotations

- Users specify functions that must be “zero alloc”
- Compiler conservatively checks the annotations

- When should this property be checked?
  - allocation is implicit in OCaml source
  - result of the check may depend on optimization

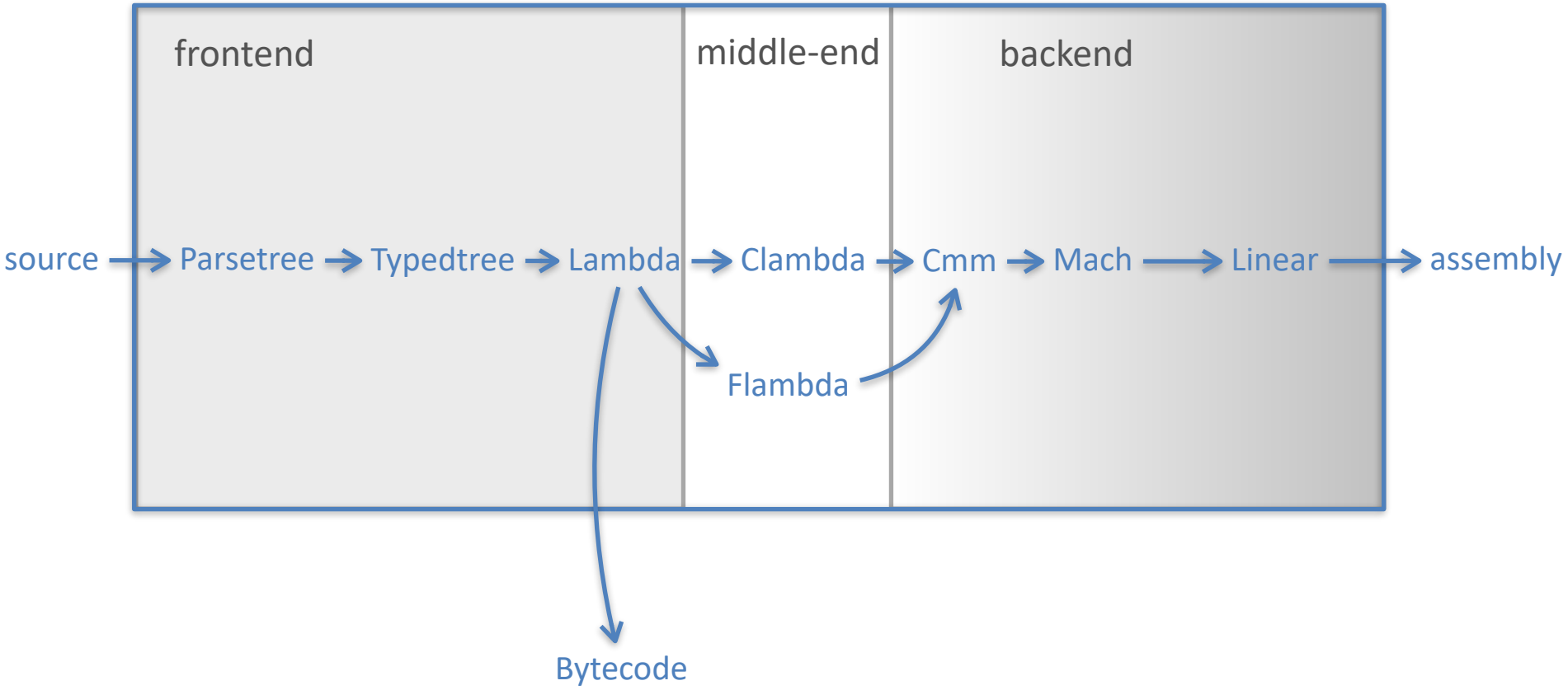
“zero alloc" annotations

- Users specify functions that must be “zero alloc”
- Compiler conservatively checks the annotations

- When should this property be checked?
  - allocation is implicit in OCaml source
  - result of the check may depend on optimization

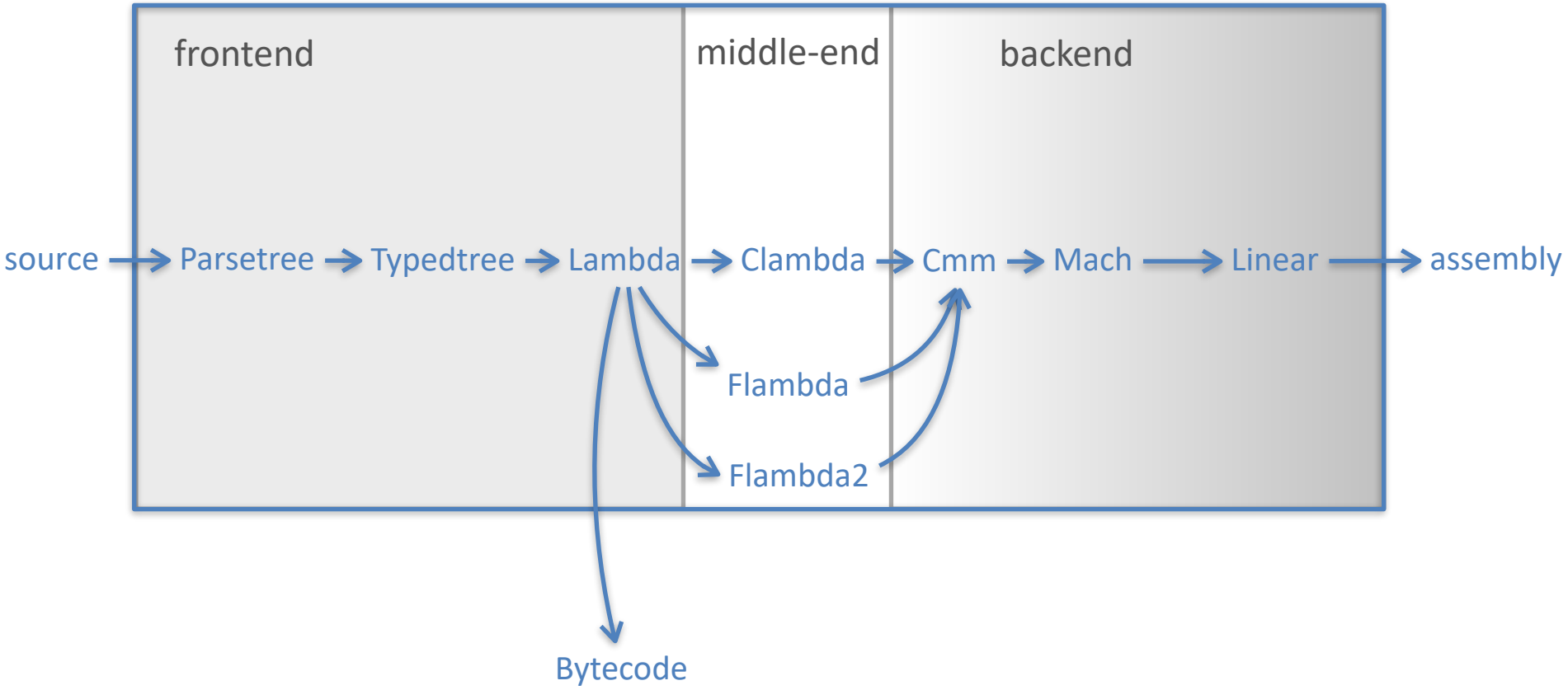
as late as possible

OCaml compiler intermediate representations



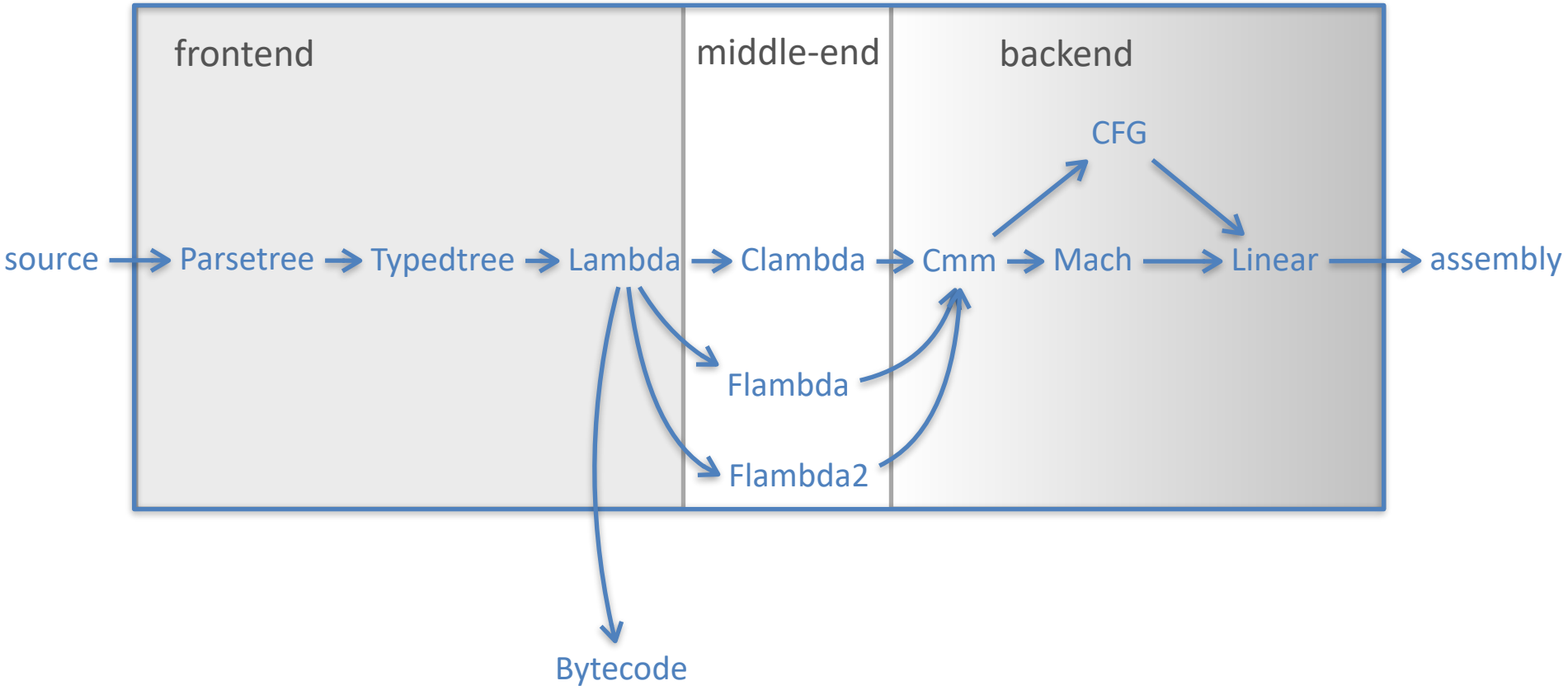
https://github.com/ocaml/ocaml

OCaml compiler intermediate representations



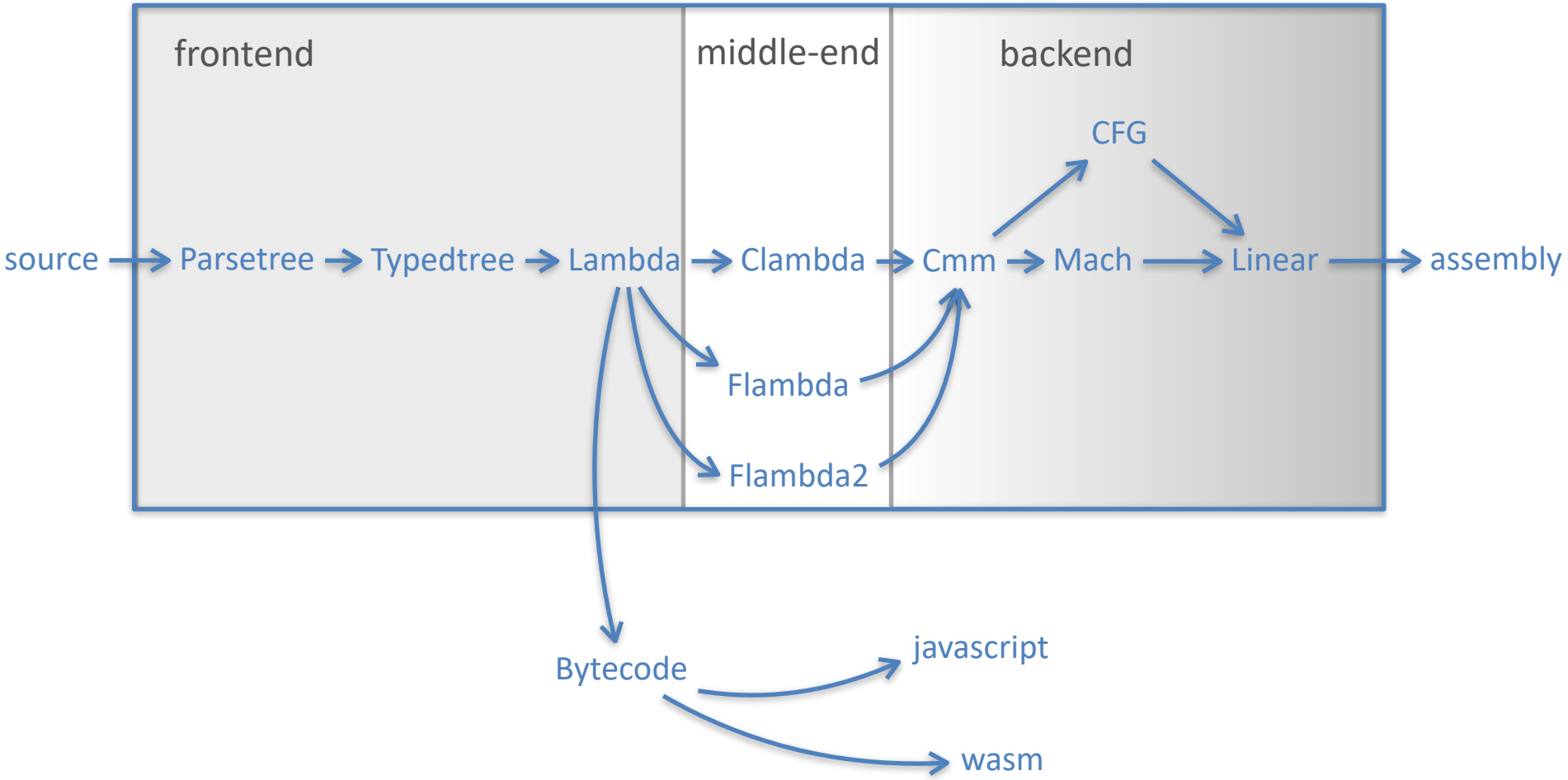
https://github.com/ocaml/ocaml

OCaml compiler intermediate representations



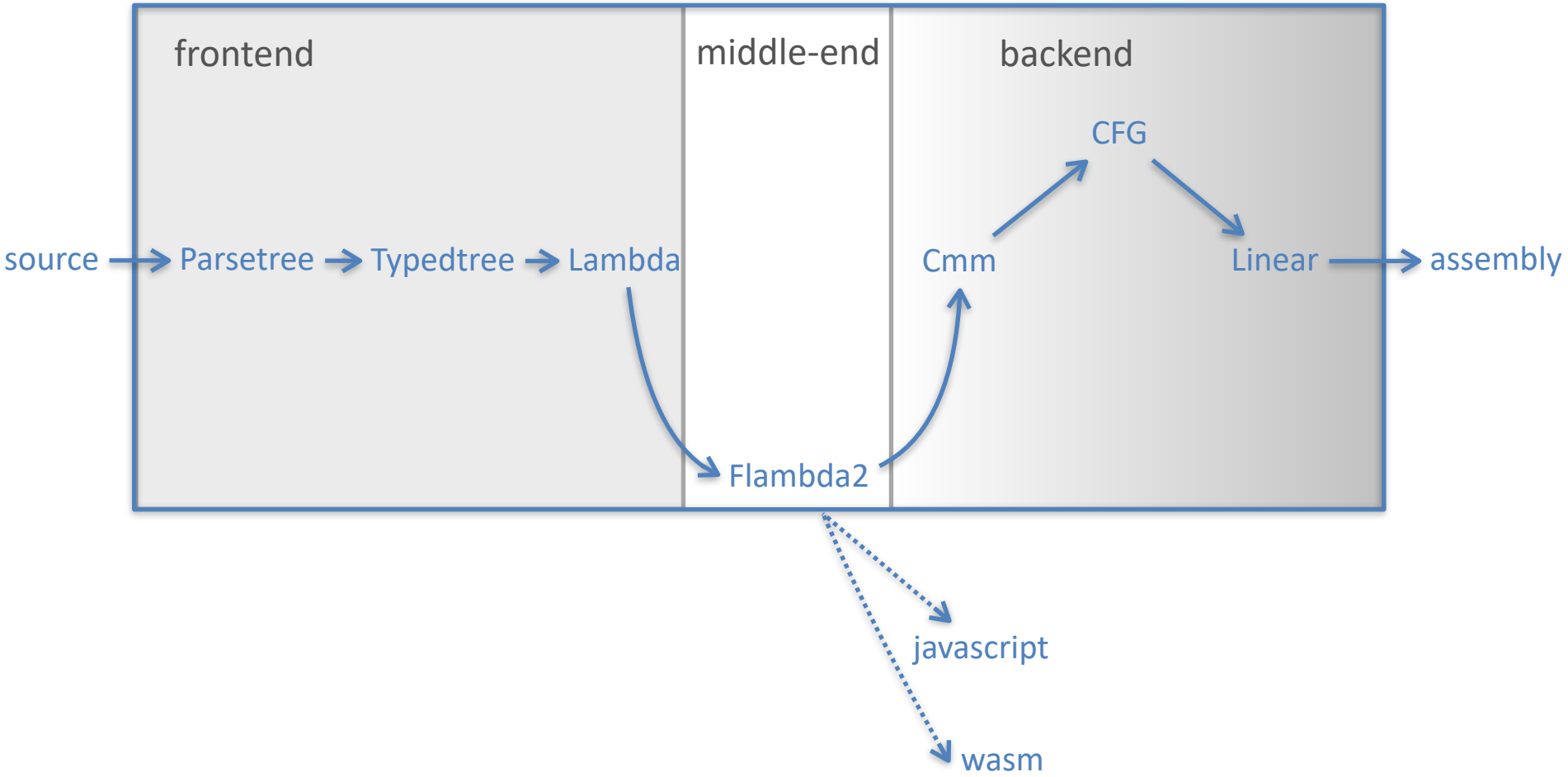
https://github.com/ocaml/ocaml

OCaml compiler intermediate representations



https://github.com/ocaml/ocaml

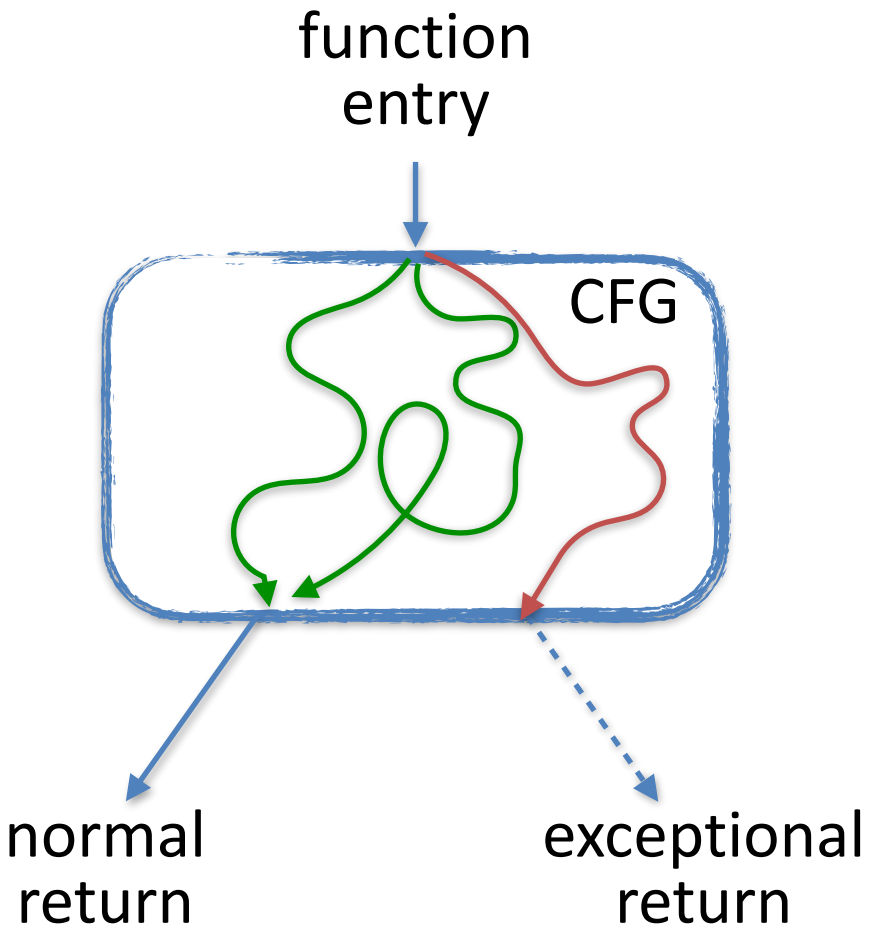
OCaml compiler intermediate representations



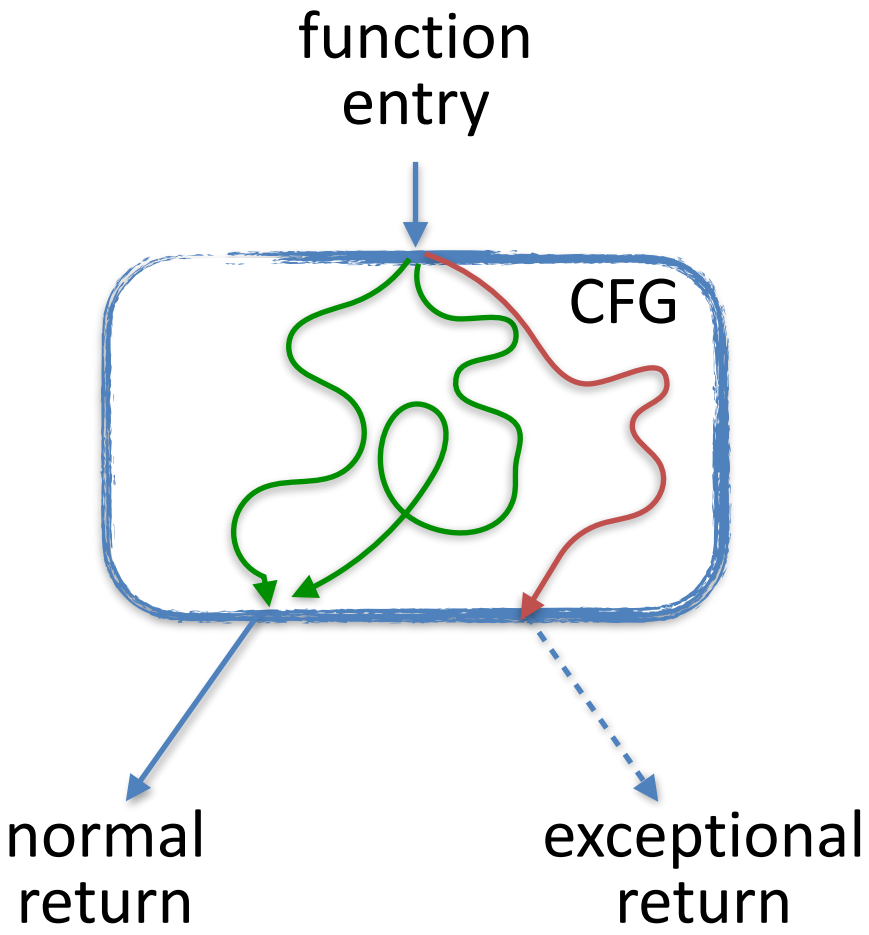
https://github.com/ocaml-flambda/flambda-backend

“zero alloc” annotations

strict vs **relaxed** semantics



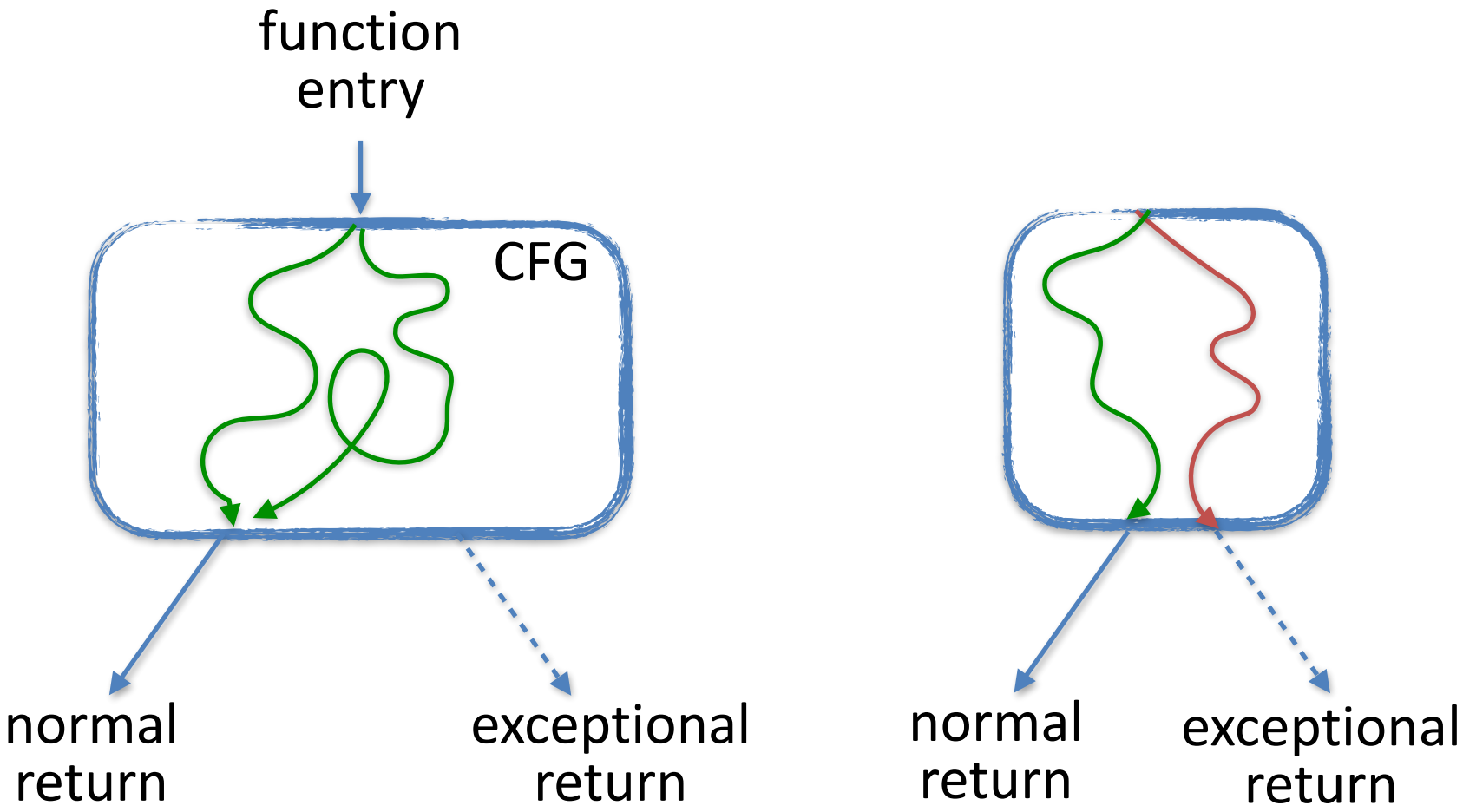
strict vs **relaxed** semantics



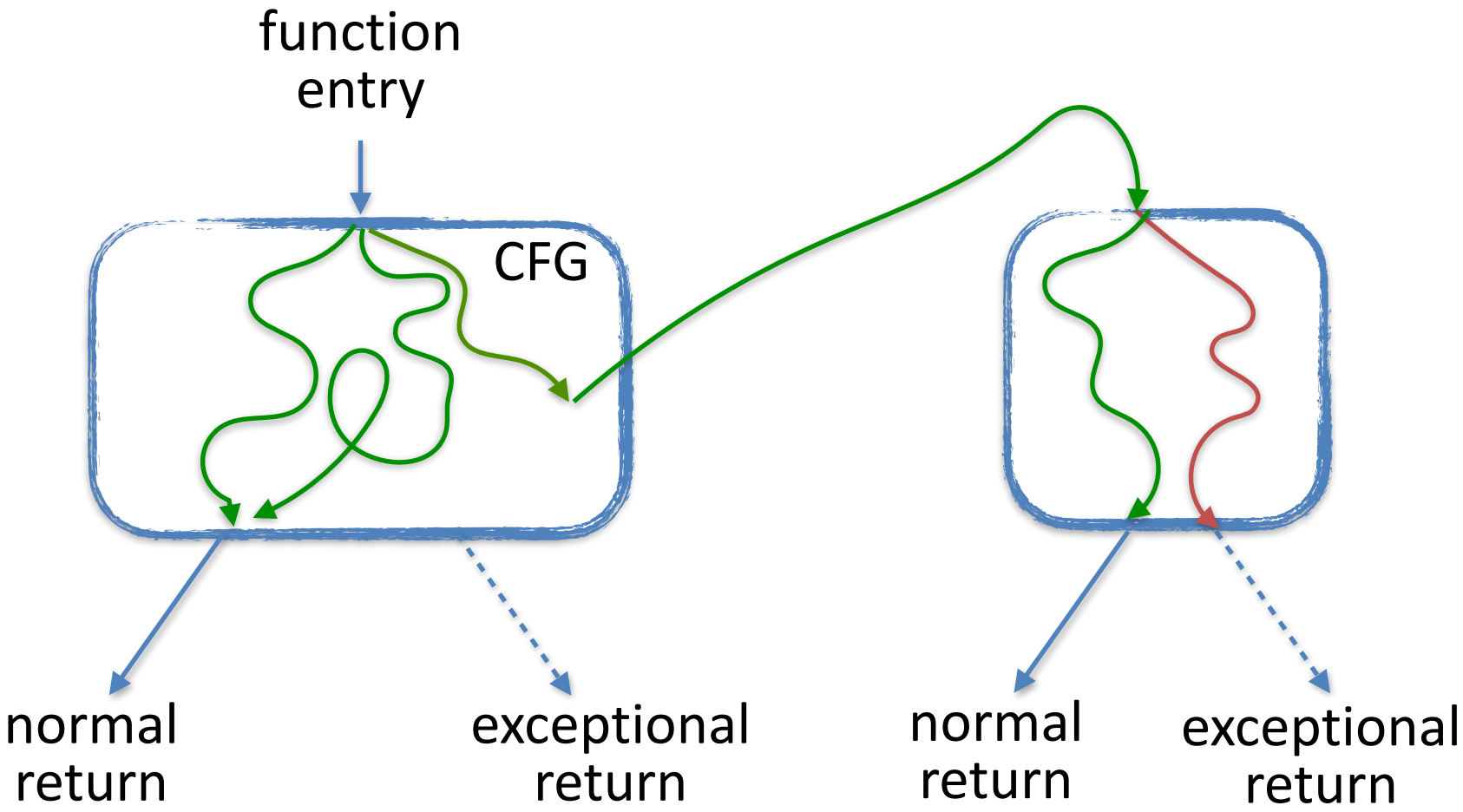
$\vDash$ [@zero_alloc]

$\nvDash$ [@zero_alloc strict]

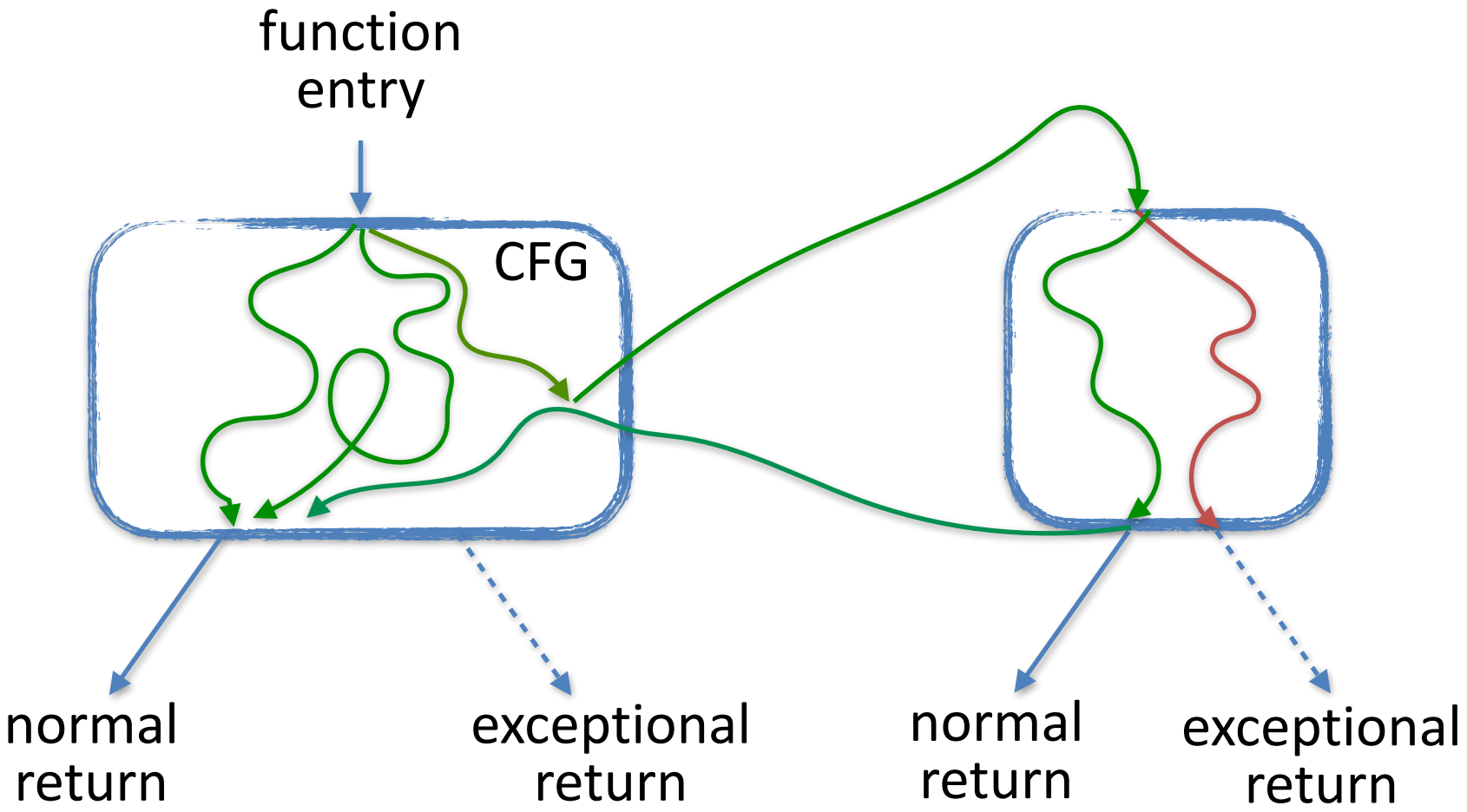
Transitive property: all callees must be zero_alloc



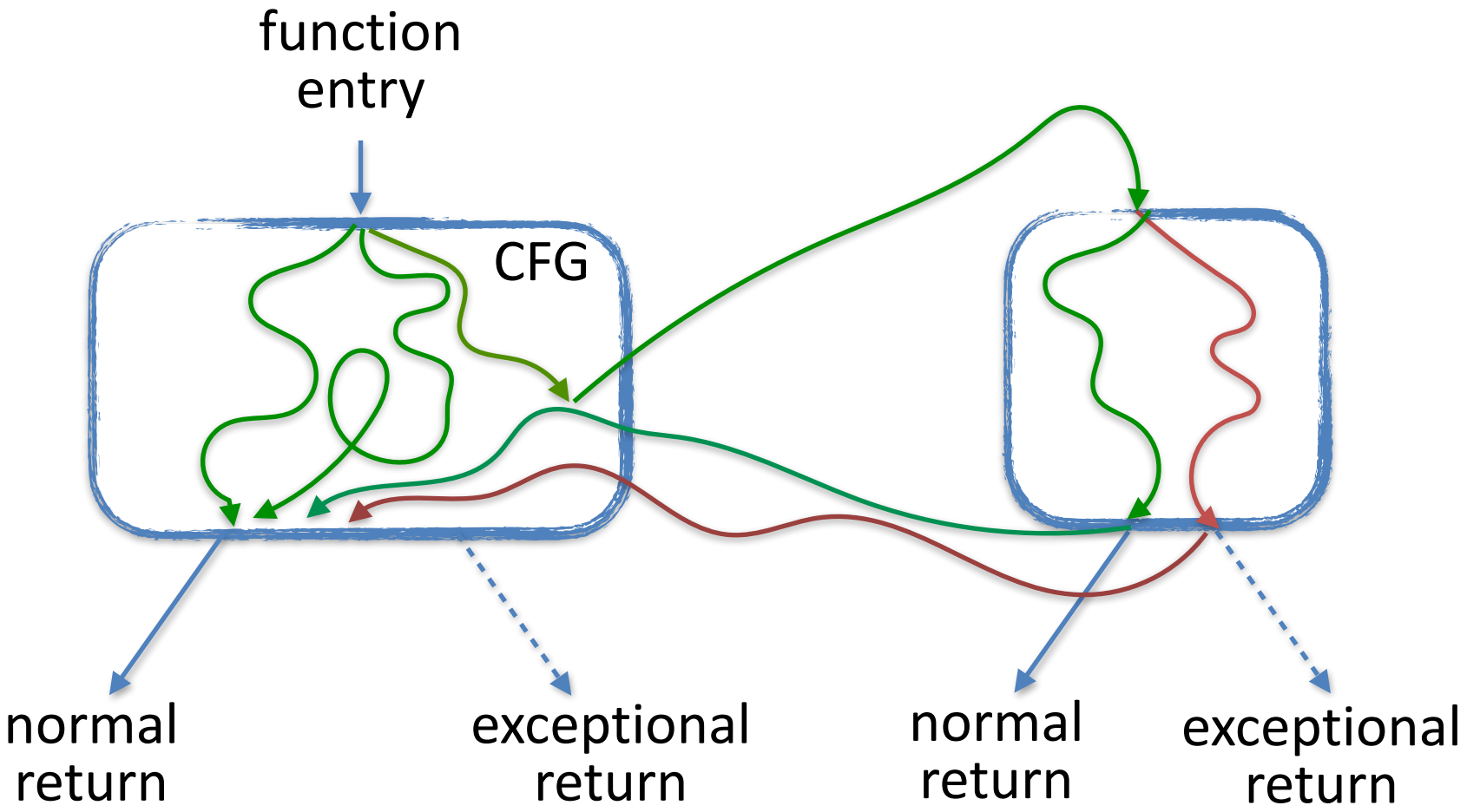
Transitive property: all callees must be zero_alloc



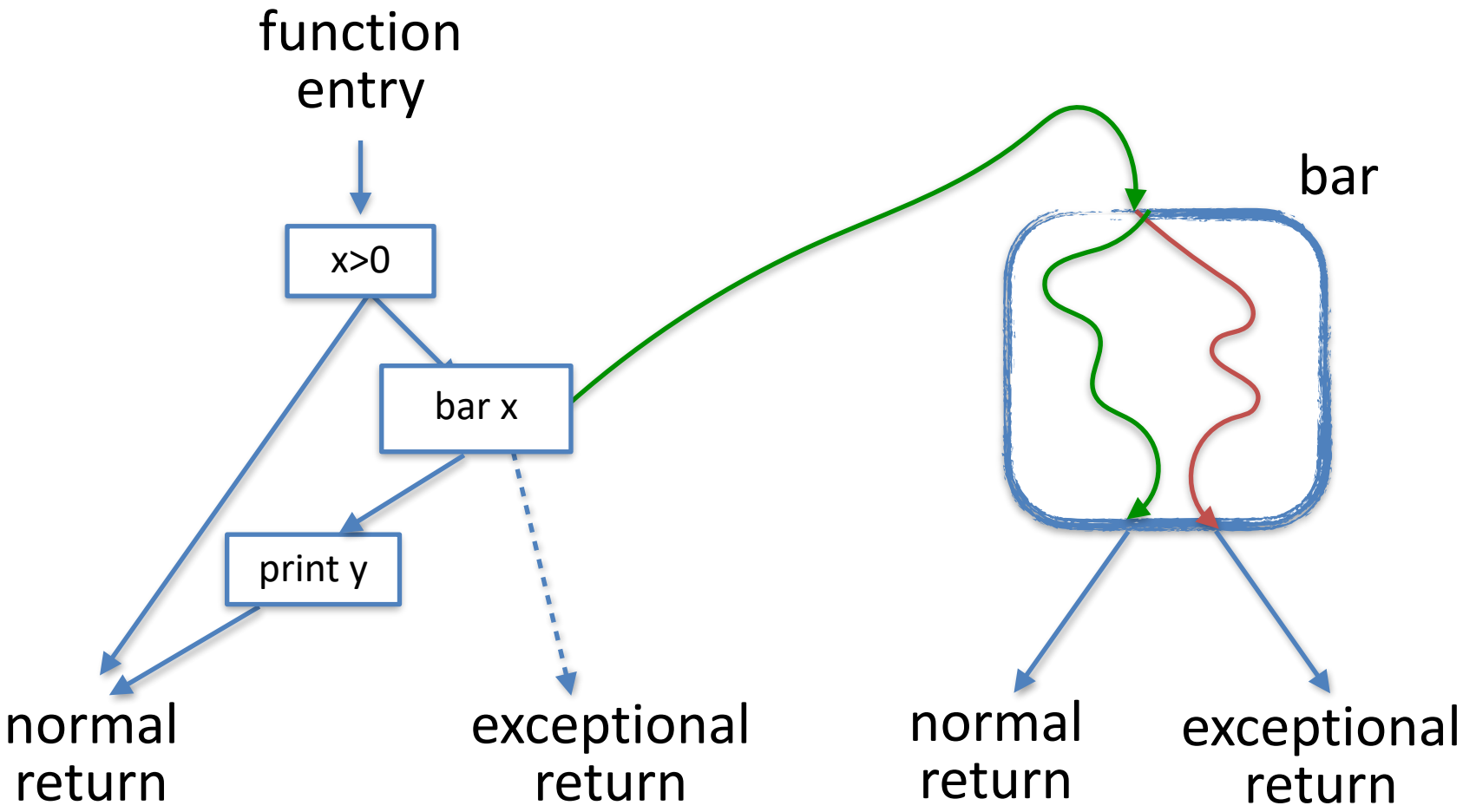
Transitive property: all callees must be zero_alloc



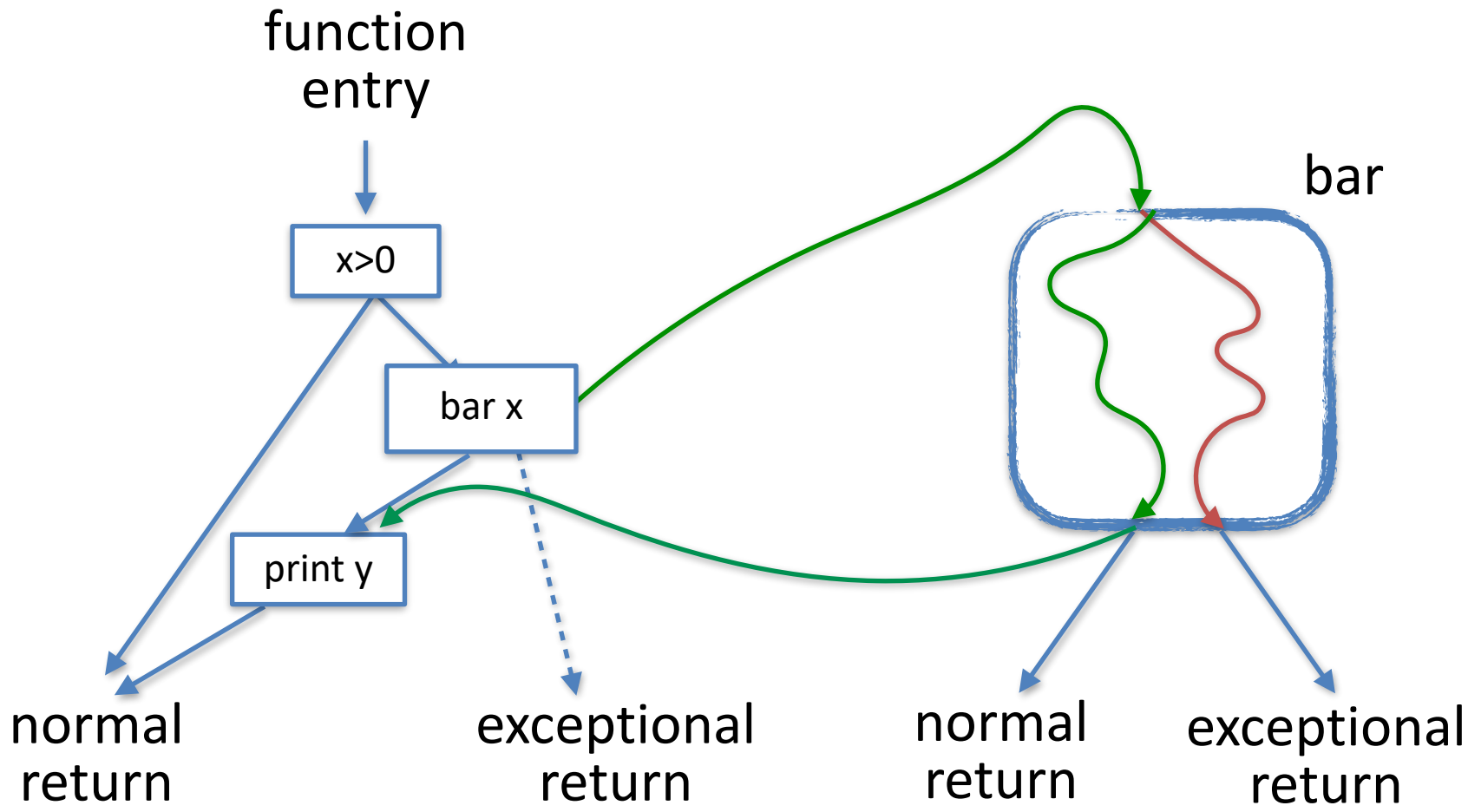
Transitive property: all callees must be zero_alloc



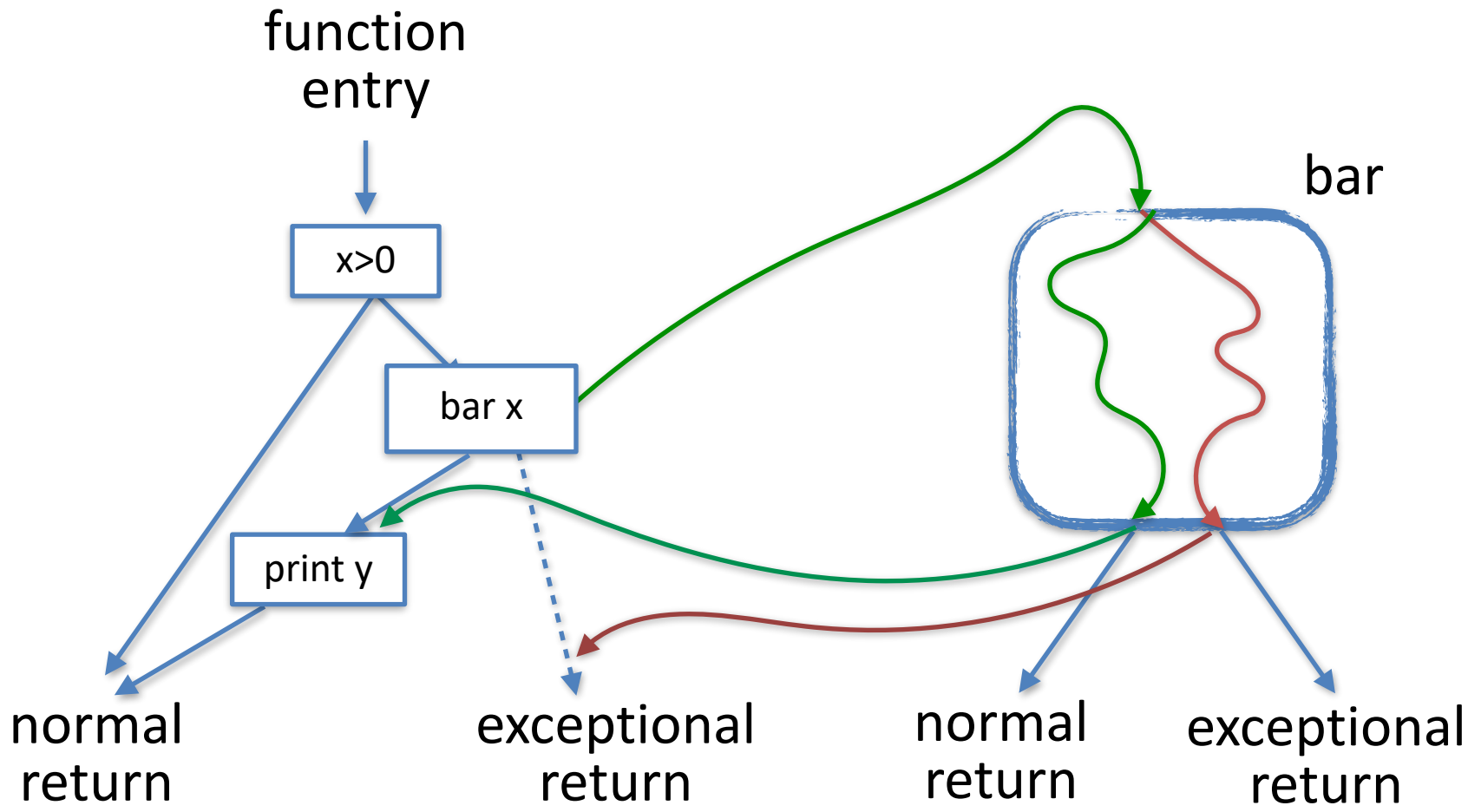
Transitive property: all callees must be zero_alloc



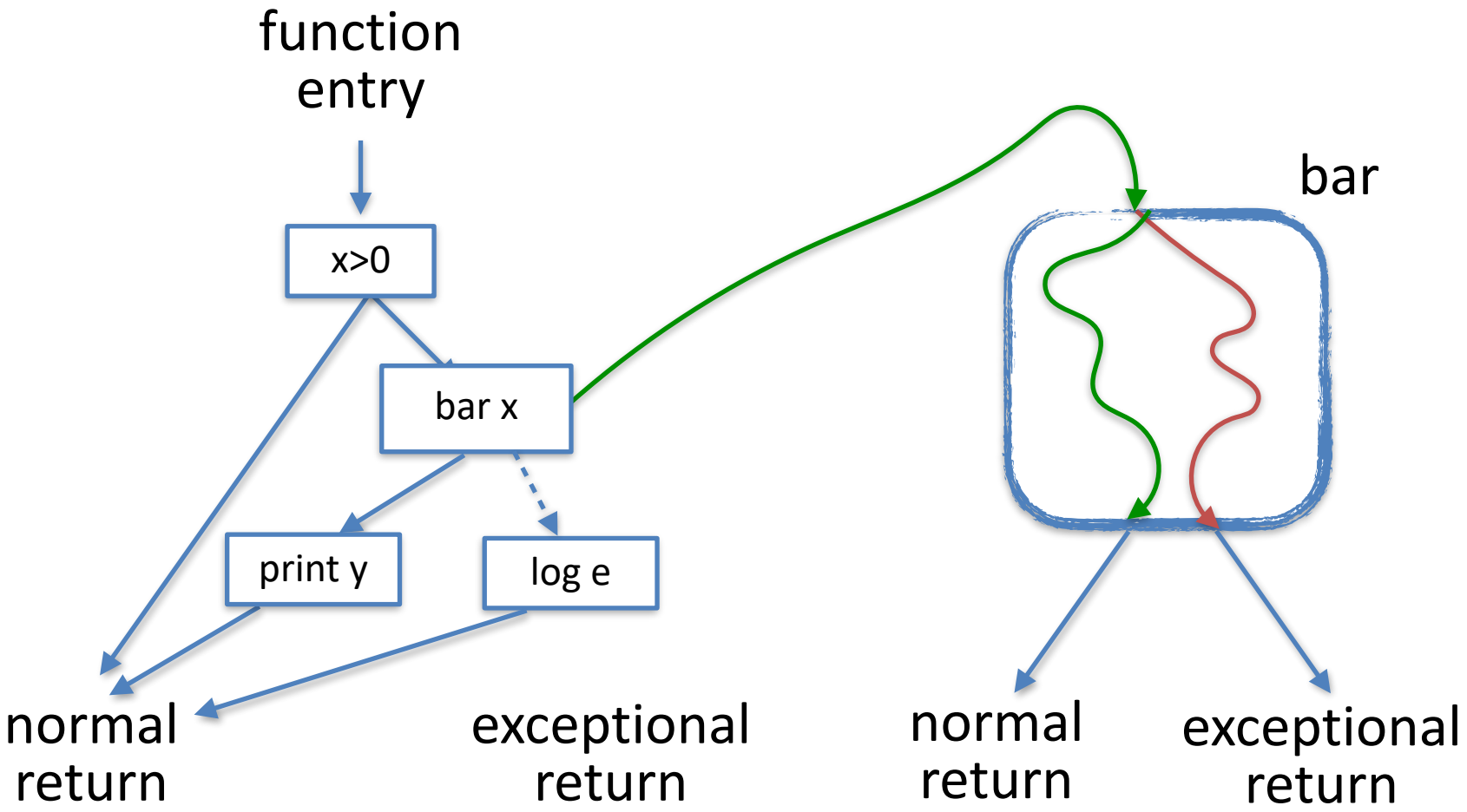
Transitive property: all callees must be zero_alloc



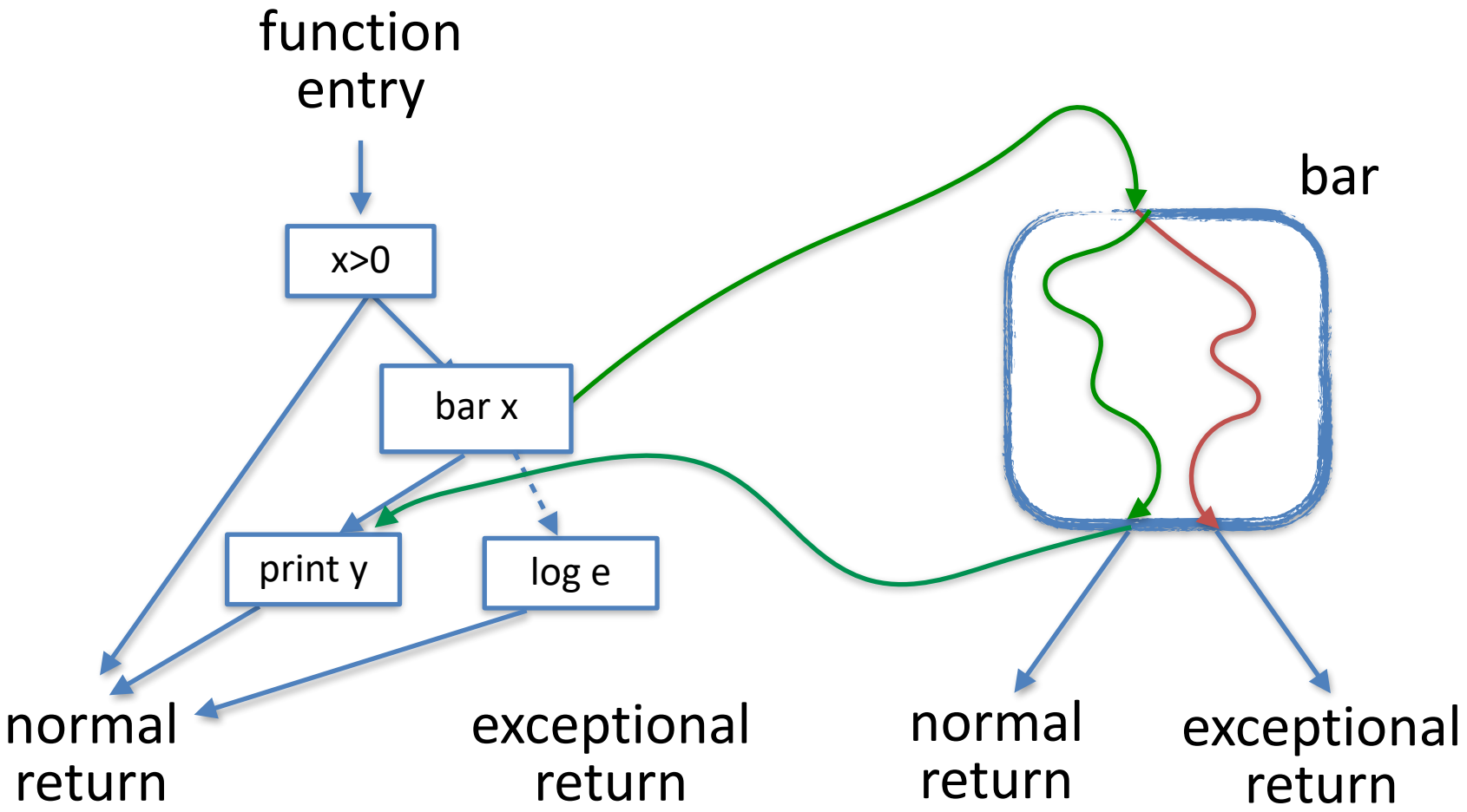
Transitive property: all callees must be zero_alloc



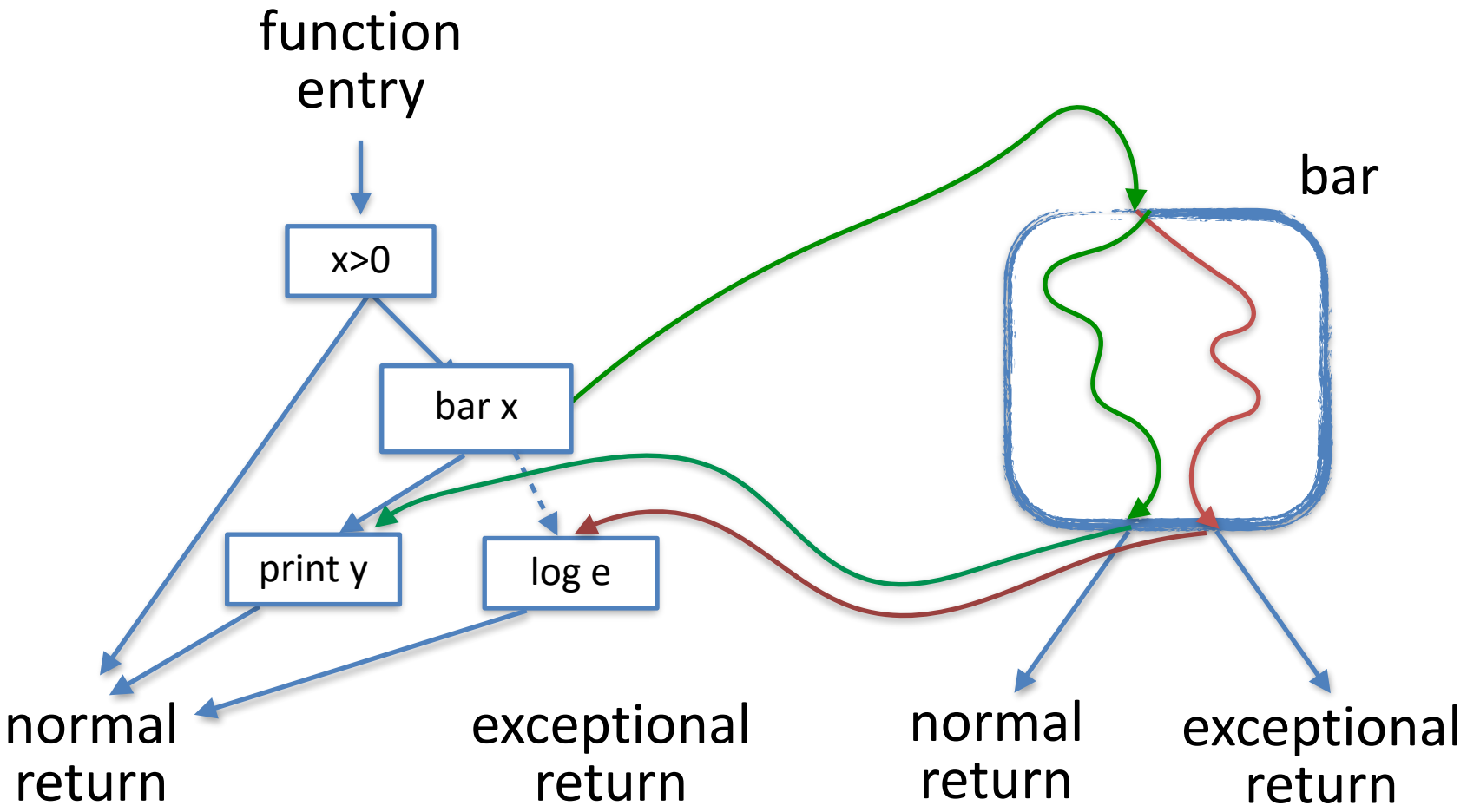
Exception handling



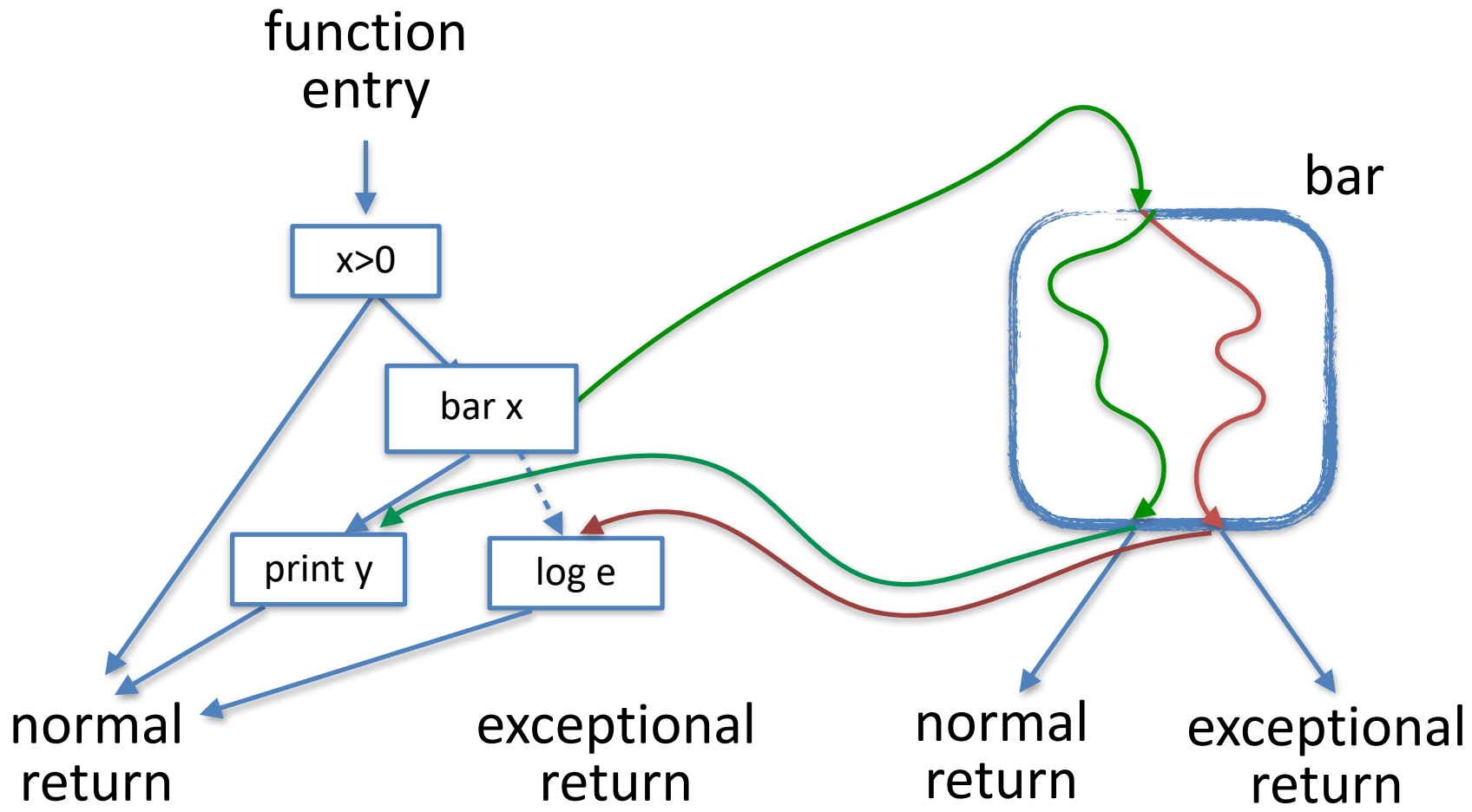
Exception handling



Exception handling



Exception handling



- Not enough to record if a function is zero_alloc
- Need to track allocation behavior on normal and exceptional return separately

Static analysis

Design requirements

- Time and memory overhead of the analysis similar to CSE and other simple backend passes
- Actionable and user-friendly error report when the check fails
- Does not affect code generation
- Sound! If the check passes, it is guaranteed that the function does not allocate a runtime
- Escape hatch for developers to control the analysis when it is overly conservative
- Failure of the check blocks merge

Abstract domain

May_allocate
|
Safe
|
Unreachable

Abstract domain

May_allocate
|
Safe
|
Unreachable

- Witness tracking: source locations of all relevant allocation sites propagated with $\top$

$$\top\ \{loc_1\} \bigsqcup \top\ \{loc_2\} = \top\ \{loc_1, loc_2\}$$

Annotation checking

Annotation checking

- Function summary is a triple `(nor,exn,div)`
  - `nor`: allocation behavior on paths from entry to normal return
  - `exn`: allocation behavior on paths from entry to exceptional return
  - `div`: allocation behavior in divergent loops reachable from entry

Annotation checking

- Function summary is a triple `(nor,exn,div)`
  - `nor`: allocation behavior on paths from entry to normal return
  - `exn`: allocation behavior on paths from entry to exceptional return
  - `div`: allocation behavior in divergent loops reachable from entry
- Annotations can be expressed in the same domain:
  - strict: `(nor=S,exn=S,div=S)`
  - relaxed: `(nor=S,exn=T,div=T)`

Annotation checking

- Function summary is a triple `(nor,exn,div)`
  - `nor`: allocation behavior on paths from entry to normal return
  - `exn`: allocation behavior on paths from entry to exceptional return
  - `div`: allocation behavior in divergent loops reachable from entry
- Annotations can be expressed in the same domain:
  - strict: `(nor=S,exn=S,div=S)`
  - relaxed: `(nor=S,exn=⊤,div=⊤)`
- Analysis computes function summaries
- Function `foo` satisfies `[@zero_alloc strict]` if `f_summary` $\sqsubseteq$ `(nor=S,exn=S,div=S)`

Abstract transformers

Abstract transformers

- Backward vs forward
- Mach vs CFG

Abstract transformers

- Backward vs forward
- Mach vs CFG
- Backward transformer for function application

$$tr(x,y) = \begin{cases} \bot & \text{if } x \text{ is } \bot \text{ or } y \text{ is } \bot \\ x \sqcup y & \text{otherwise} \end{cases}$$

Abstract transformers

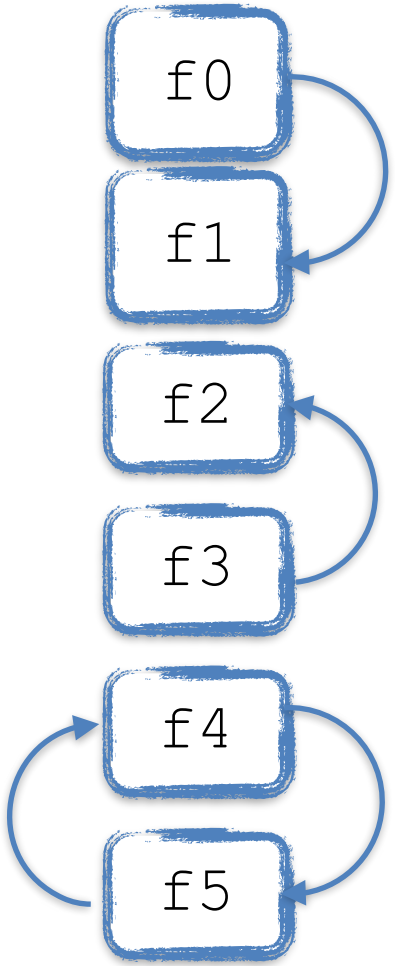
- Backward vs forward
- Mach vs CFG
- Backward transformer for function application
  - applied pointwise to `(nor,exn,div)`
  - commutative and associative

$$\mathrm{tr}(x,y) = \begin{cases} \bot & \text{if } x \text{ is } \bot \text{ or } y \text{ is } \bot \\ x \sqcup y & \text{otherwise} \end{cases}$$

Abstract transformers

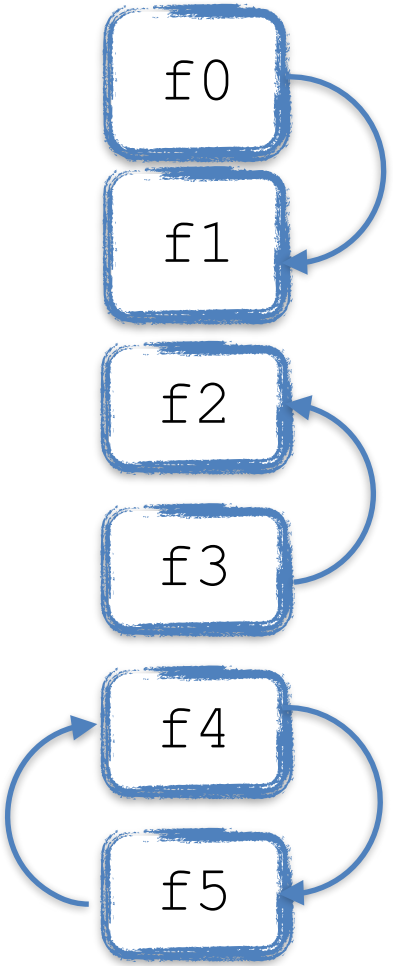
- Identity except..
- Allocation: treat as a function application with summary `(nor=⊤,exn=⊥,div=⊥)`
- Indirect calls: conservatively assume summary is `(nor=⊤,exn=⊥,div=⊥)`
- External calls may allocate
  - unless annotated `[@noalloc]`
  - but not the same as `[@zero_alloc]`
  - `[@noalloc]` affects code generation
  - `[@noalloc]` does not have relaxed meaning

Order of functions matters



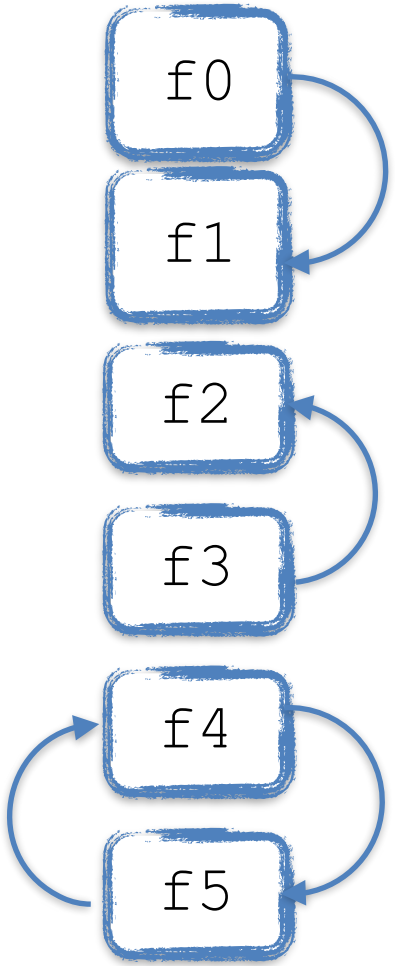
Order of functions matters

- The backend compiles one function at a time all the way from Cmm to Assembly
- How to handle forward dependencies?
- How to check recursive functions?



Order of functions matters

- The backend compiles one function at a time all the way from Cmm to Assembly
- How to handle forward dependencies?
- How to check recursive functions?
- Conservative
  - order of functions in a compilation unit affects precision of analysis results
- Hold on to the CFG until all of its dependencies are resolved
  - increases memory footprint
  - CFG is mutable
- Hold on to a “mini-CFG”



Symbolic domain

Symbolic domain

- Abstract values are constraints in normal form
  - Constant: ⊤ S ⊥
  - Variable: represents a component of a function summary of an unresolved dependency
  - Transform (a1,..an) where argument is either a variable or ⊤
  - Join(b1,…,bm) where argument is either a variable, or a transform, or ⊤ or S

Symbolic domain

- Abstract values are constraints in normal form
  - Constant: ⊤ S ⊥
  - Variable: represents a component of a function summary of an unresolved dependency
  - Transform (a1,..an) where argument is either a variable or ⊤
  - Join(b1,…,bm) where argument is either a variable, or a transform, or ⊤ or S
- Finite height
- Normalization is exponential: distributing join over transform
  - tr(x ⊔ y , z) —> tr(x,z) ⊔ tr(y,z)
  - intuition: path constraints
- Naive implementation
- Heuristics
  - bounded witnesses
  - bounded join

Handling functions defined in other compilation units

- Problem: separate compilation
- How can module A know if a function in module B allocates?
- Cross-module inlining already has access to IR of the function (cmx file)
- Add allocation summaries to the same compilation artifact

Source of false alarms

- module implementation not available
  - indirect function call
  - build system “hides” dependencies
- correlations
- error values that are not exceptions

Escape hatch

`[@zero_alloc assume]`

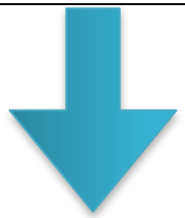
- Annotation on function definition or application
- Static analysis can use it as function summary
- How to propagate it to the backend?

Assume and inlining

```
let[@zero_alloc assume] bar x =
  if x > 0 then f x
  else (x,x+1)
let foo x = bar x
```

Assume and inlining

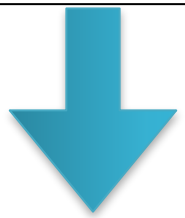
```
let[@zero_alloc assume] bar x =
  if x > 0 then f x
  else (x,x+1)
let foo x = bar x
```



```
let[@zero_alloc assume] bar x =
  if x > 0 then (f[@zero_alloc assume]) x
  else ((x,x+1)[@zero_alloc assume])
let foo x = bar x
```

Assume and inlining

```
let[@zero_alloc assume] bar x =
  if x > 0 then f x
  else (x,x+1)
let foo x = bar x
```



```
let[@zero_alloc assume] bar x =
  if x > 0 then (f[@zero_alloc assume]) x
  else ((x,x+1)[@zero_alloc assume])
let foo x = bar x
```

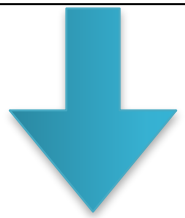
- transformation in lambda to mark all “primitives” as zero_alloc
- piggy back on debug info to propagate to backend

Assume and exception handling

```
let[@zero_alloc assume] bar x =
  try f x
  with e -> h e
let foo x = bar x
```

Assume and exception handling

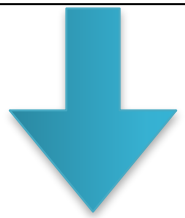
```
let[@zero_alloc assume] bar x =
  try f x
  with e -> h e
let foo x = bar x
```



```
let[@zero_alloc assume] bar x =
  try (f[@zero_alloc assume]) x
  with e -> (h[@zero_alloc assume]) e
let foo x = bar x
```

Assume and exception handling

```
let[@zero_alloc assume] bar x =
  try f x
  with e -> h e
let foo x = bar x
```



```
let[@zero_alloc assume] bar x =
  try (f[@zero_alloc assume]) x
  with e -> (h[@zero_alloc assume]) e
let foo x = bar x
```

- check of foo fails if bar is inlined

Give user more control of “assume”

Give user more control of “assume”

```
let[@zero_alloc] bar x =
  try (f[@zero_alloc assume strict]) x
  with e -> h e
let foo x = bar x
```

```
let[@zero_alloc] bar x =
  try f x
  with e -> (h[@zero_alloc assume never_returns_normally]) e
let foo x = bar x
```

```
let[@zero_alloc] bar x =
  try f x
  with e -> (h[@zero_alloc error]) e
let foo x = bar x
```

Give user more control of “assume”

```
let[@zero_alloc] bar x =
  try (f[@zero_alloc assume strict]) x
  with e -> h e
let foo x = bar x
```

```
let[@zero_alloc] bar x =
  try f x
  with e -> (h[@zero_alloc assume never_returns_normally]) e
let foo x = bar x
```

```
let[@zero_alloc] bar x =
  try f x
  with e -> (h[@zero_alloc error]) e
let foo x = bar x
```

- solution: track exception scope for each allocation

zero_alloc in signatures

Functors

```
module type T = sig
  val foo : int -> int
end

module F(S:T) = struct
  let bar x = S.foo x
end
```

- How can functor F know if a function in its argument S allocates?
- Should we track zero_alloc as part of a function type?
- Lightweight alternative: track zero_alloc in module types

Functors

```
module type T = sig
  val[@zero_alloc] foo : int -> int
end

module F(S:T) = struct
  let[@zero_alloc] bar x = S.foo x
end
```

- How can functor F know if a function in its argument S allocates?
- Should we track zero_alloc as part of a function type?
- Lightweight alternative: track zero_alloc in module types

Handle zero_alloc in signatures

A.mli

```
val[@zero_alloc] foo :
  int -> int -> int
```

B.ml

```
let[@zero_alloc] bar x =
  if x < 100 then
    A.foo x (x + 1)
  else 0
```

Handle zero_alloc in signatures

A.mli

```
val[@zero_alloc] foo :
  int -> int -> int
```

B.ml

```
let[@zero_alloc] bar x =
  if x < 100 then
    A.foo x (x + 1)
  else 0
```

- Typechecker knows that `foo` is zero_alloc
- How to communicate it to the backend?

Handle zero_alloc in signatures

A.mli

```
val[@zero_alloc] foo :
  int -> int -> int
```

B.ml

```
let[@zero_alloc] bar x =
  if x < 100 then
    A.foo x (x + 1)
  else 0
```

- Typechecker knows that `foo` is zero_alloc
- How to communicate it to the backend?
- Add **assume** annotation on application

Handle zero_alloc in signatures

A.mli

```
val[@zero_alloc] foo :
  int -> int -> int
```

B.ml

```
let[@zero_alloc] bar x =
  if x < 100 then
    A.foo x (x + 1)
  else 0
```

- Typechecker knows that `foo` is zero_alloc
- How to communicate it to the backend?
- Add **assume** annotation on application

B.ml (after typing)

```
let[@zero_alloc] bar x =
  if x < 100 then
    (A.foo[@zero_alloc assume]) x (x + 1)
  else 0
```

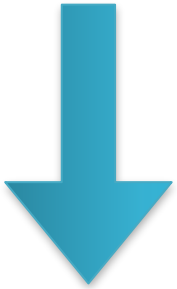
Infer zero_alloc from signatures

A.mli

```
val[@zero_alloc] foo :
  int -> int -> int
```

A.ml

```
let foo x y =
  x+y
```



A.ml

```
let[zero_alloc] foo x y =
  x+y
```

B.ml

```
let[@zero_alloc] bar x =
  if x < 100 then
    A.foo x (x + 1)
  else 0
```

B.ml (after typing)

```
let[@zero_alloc] bar x =
  if x < 100 then
    (A.foo[@zero_alloc assume]) x (x + 1)
  else 0
```

Problem: arity

A.mli

```
val[@zero_alloc] foo :
  int -> int -> int
```

B.ml

```
let[@zero_alloc] bar x =
 A.foo (x + 1)
```

Problem: arity

A.mli

```
val[@zero_alloc] foo :
  int -> int -> int
```

B.ml

```
let[@zero_alloc] bar x =
 A.foo (x + 1)
```

- Adding assume is not sound!
  - bar allocates a closure
- Solution:
  - zero_alloc in signatures also tracks arity
  - applications only get "**assume**" if fully applied

Problem: separate compilation

- Enable checking “zero alloc” in fast build using “zero alloc” in signatures
- Fast builds
  - Developer’s local builds for interactive editing
  - Continuous integration
- Optimizing builds (slow)
  - benchmarking
  - release and deployment in production

Problem: allocating only in fast build

- Optimization may be needed to eliminate allocations
  - static allocation
  - unboxing
  - resolving calls
- Solution: users specifies functions that can only be checked in optimizing builds `[@zero_alloc opt]`
- How does the compiler know when to check?
- Can we treat these functions as “zero alloc” in fast build?

Open question: higher-order functions

```
let rec iter f l =
  match l with
  | [] -> ()
  | x :: xs -> f x; iter f xs
```

Open question: higher-order functions

```
let rec iter f l =
  match l with
  | [] -> ()
  | x :: xs -> f x; iter f xs
```

- `iter f l` is zero alloc if `f` is zero alloc

Open question: higher-order functions

```
let rec iter f l =
  match l with
  | [] -> ()
  | x :: xs -> f x; iter f xs
```

- `iter f l` is zero alloc if `f` is zero alloc
- Can we add some polymorphism?

  let[@zero_alloc 'z] **rec** iter (f[@zero_alloc 'z]) l = ..

Open question: higher-order functions

```
let rec iter f l =
  match l with
  | [] -> ()
  | x :: xs -> f x; iter f xs
```

- `iter f l` is zero alloc if `f` is zero alloc
- Can we add some polymorphism?

  let[@zero_alloc 'z] **rec** iter (f[@zero_alloc 'z]) l = ..

- Idea: check body under assumption `f` is zero_alloc

Open question: higher-order functions

```
let rec iter f l =
  match l with
  | [] -> ()
  | x :: xs -> f x; iter f xs
```

- `iter f l` is zero alloc if `f` is zero alloc
- Can we add some polymorphism?

  let[@zero_alloc 'z] **rec** iter (f[@zero_alloc 'z]) l = ..

- Idea: check body under assumption `f` is zero_alloc
  - unsound
  - zero_alloc assumption can escape

Current limitations

- Higher-order functions
- Interaction of assume on try-with with inlining
- Scoping of zero_alloc annotations
- Annotations on sub-expressions
- Interaction with dead code elimination
  - how to check functions that are eliminated?
- Aliases fail the check

```
let foo x = x + 1
let[@zero_alloc] bar = foo
```

Byproduct of “zero alloc” analysis

- Computes a conservative approximation of other function properties
  - may raise
  - does not return normally
  - contain indirect calls

Takeaways

- “front-end feature describing a back-end property”
- Simple static analysis
- Actionable report when the check fails
- Reduce annotation burden but give users enough control
- Workflow integration: fast interactive builds matter!
- Huge effort annotating existing code
- Not a replacement for dynamic checking

Team work on “zero alloc”

Main dev and review by

+ Chris Casinghino

+ Xavier Clerc

+ Leo White

+ Greta Yorsh

In collaboration with others from

+ OCaml Language Team

+ Build System Team

+ Editor Integration Team

+USERS

Writing performance-sensitive code in OCaml

- Experiment with language features
  - unboxed types
  - local allocations
  - data-race freedom
- Compiler optimizations
  - inlining
  - unboxing
  - feedback-directed code layout
  - register allocation
  - vectorizer
  - prelinking
- Tracing, profiling, and debugging
  - memtrace
  - magic trace
  - ocaml-probes

https://github.com/ocaml-flambda/flambda-backend

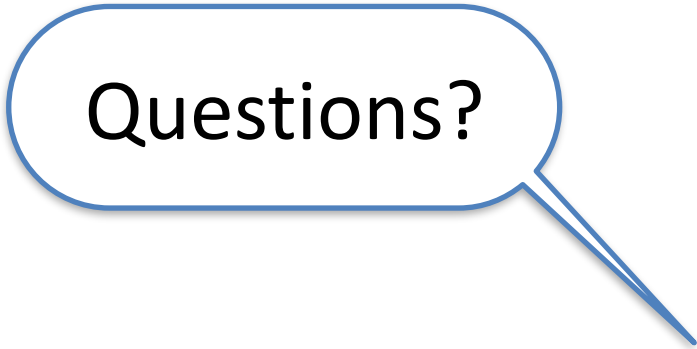
https://www.janestreet.com/

https://github.com/janestreet

THANK YOU

Jane Street

THANK YOU



Jane Street