Stack allocation for OCaml

Stephen Dolan, Leo White

Jane Street

Allocation

- Frequent

```
type node = {
  name: string;
  successors: string list
}

type graph = node list

let count_self_edges (g : graph) =
  let count = ref 0 in
  List.iter
    (fun node ->
      List.iter
        (fun succ ->
          if succ = node.name then incr count)
        node.successors)
    g;
  !count
```

Allocation

- Frequent

```
type node = {
  name: string;
  successors: string list
}

type graph = node list

let count_self_edges (g : graph) =
  let count = ref 0 in
  List.iter
    (fun node ->
      List.iter
        (fun succ ->
          if succ = node.name then incr count)
        node.successors)
    g;
  !count
```

- 16 bytes for the `ref`

Allocation

- Frequent

```
type node = {
  name: string;
  successors: string list
}

type graph = node list

let count_self_edges (g : graph) =
  let count = ref 0 in
  List.iter
    (fun node ->
      List.iter
        (fun succ ->
          if succ = node.name then incr count)
        node.successors)
    g;
  !count
```

- 16 bytes for the `ref`
- 32 bytes for the `fun node -> ...` closure

Allocation

- Frequent

```
type node = {
  name: string;
  successors: string list
}

type graph = node list

let count_self_edges (g : graph) =
  let count = ref 0 in
  List.iter
    (fun node ->
      List.iter
        (fun succ ->
          if succ = node.name then incr count)
        node.successors)
    g;
  !count
```

- 16 bytes for the `ref`
- 32 bytes for the `fun node -> ...` closure
- 40×N bytes for the `fun succ -> ...` closure

Allocation

Short-lived allocations are cheap, but:

- Frequent
- Cheap, not free

Allocation

- Frequent
- **Cheap, not free**

Short-lived allocations are cheap, but:

- Space is not reused quickly
  causing poor L1 cache usage
- GC advances towards the next minor GC
  so other allocations are promoted unnecessarily

Allocation

- Frequent
- Cheap, not free
- Stack allocation

Allocating values on a stack:

- reuses space quickly
- does not cause any GC work

Allocation

- Frequent
- Cheap, not free
- Stack allocation

Allocating values on a stack:

- reuses space quickly
- does not cause any GC work

Hard to do safely, though!

When is it safe to pass
stack-allocated values
to a function?

Prior work

Region variables attach lifetime information to types.

- Region variables

Prior work

- Region variables

Region variables attach lifetime information to types.

```
struct TwoBorrowedStrings<'a> {
    x: &'a str,
    y: &'a str,
}
```

Prior work

- Region variables

Region variables attach lifetime information to types.

```
struct TwoBorrowedStrings<'a> {
    x: &'a str,
    y: &'a str,
}
```

- Extremely expressive
- Syntactically heavyweight

Prior work

- Region variables
- Stack arguments

Functions that can accept stack arguments are typed with **region polymorphism**:

$$\forall\alpha.\ \texttt{TwoBorrowedStrings}[\alpha] \rightarrow ()$$

Prior work

- Region variables
- Stack arguments

Functions that can accept stack arguments are typed with **region polymorphism**:

$$\forall\alpha.\ \texttt{TwoBorrowedStrings}[\alpha] \rightarrow ()$$

Higher-order functions can require higher-rank (non-inferrable) types.

Modes, not types

- Local and global

Instead, we mark variable bindings as `local` or `global`:

- `global` bindings never refer to stack-allocated values
- `local` bindings never escape their *region*
  (function body or loop)

Modes, not types

- Local and global

Instead, we mark variable bindings as `local` or `global`:

- `global` bindings never refer to stack-allocated values
- `local` bindings never escape their *region*
  (function body or loop)

Less expressive than region variables, but much simpler.

Modes, not types

- Local and global
- Modes are deep

The same types are used at `local` and `global` mode:

```
type node = {
  name: string;
  successors: string list
}

type graph = node list
```

A local `graph` has local contents.

Modes, not types

- Local and global
- Modes are deep

The same types are used at `local` and `global` mode:

```
type node = {
  name: string;
  successors: string list
}

type graph = node list
```

A local `graph` has local contents.

```
type part_global = {
  foo : string;
  global_ bar : string;
}
```

`(...).bar` is always global.

Modes, not types

- Local and global
- Modes are deep
- Function types

Our function types specify the mode of their argument:

```
type s = string -> unit
type t = local_ string -> unit
```

Modes, not types

- Local and global
- Modes are deep
- **Function types**

Our function types specify the mode of their argument:

```
type s = string -> unit
type t = local_ string -> unit
```

A function of type `local_ 'a -> 'b` cannot capture its argument, so can be passed a stack-allocated value.

Modes, not types

- Local and global
- Modes are deep
- **Function types**

Our function types specify the mode of their argument:

```
type s = string -> unit
type t = local_ string -> unit
```

A function of type `local_ 'a -> 'b` cannot capture its argument, so can be passed a stack-allocated value.

No lifetime variables or polymorphism, so inference works.

Modes, not types

- Local and global
- Modes are deep
- Function types
- **Local returns**

Function types also have a mode on the return type:

```
module M : sig
  val f : 'a -> local_ 'a option
end = struct
  let f x = local_ (Some x)
end
```

Modes, not types

- Local and global
- Modes are deep
- Function types
- **Local returns**

Function types also have a mode on the return type:

```
module M : sig
  val f : 'a -> local_ 'a option
end = struct
  let f x = local_ (Some x)
end
```

Separating the data from the control stack means values can be allocated in the caller's region.

Modes, not types

- Local and global
- Modes are deep
- Function types
- Local returns
- Typing closures

Typing rule for closures:

$$\frac{\Gamma, \quad x : A \vdash e : B}{\Gamma \vdash \texttt{fun}\ x \to e : \quad A \to \quad B}$$

Modes, not types

- Local and global
- Modes are deep
- Function types
- Local returns
- Typing closures

Typing rule for closures with modes:

$$\frac{\Gamma,\ \square_i,\ j\ x : A \vdash e : B \quad @\ k}{\Gamma \vdash \mathtt{\textbf{fun}}\ x \to e : j\ A \to k\ B \quad @\ i}$$

Modes, not types

- Local and global
- Modes are deep
- Function types
- Local returns
- Typing closures

Typing rule for closures with modes:

$$\frac{\Gamma,\ \square_i,\ j\ x : A \vdash e : B\ \ @\ k}{\Gamma \vdash \mathtt{fun}\ x \to e : j\ A \to k\ B\ \ @\ i}$$

- i: mode of the closure itself
- j: mode of the argument
- k: mode of the return

Modes, not types

- Local and global
- Modes are deep
- Function types
- Local returns
- Typing closures

Typing rule for closures with modes:

$$\frac{\Gamma,\ \square_i,\ j\ x : A \vdash e : B \ \ @\ k}{\Gamma \vdash \texttt{fun}\ x \rightarrow e : j\ A \rightarrow k\ B \ \ @\ i}$$

- i: mode of the closure itself
- j: mode of the argument
- k: mode of the return

Variable access must agree with locking:

$$\frac{i \leq j \qquad i \leq k}{\Gamma,\ ix : A,\ \ldots,\ \square_j\ , \ldots\ \vdash x : A \ \ @\ k}$$

where `global` $\leq$ `local`.

Examples

- Iteration

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

Examples

- Iteration

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

```
let count_self_edges (g : graph) =
  let count = ref 0 in
  List.iter
    (fun node ->
      List.iter
        (fun succ ->
          if succ = node.name then incr count)
        node.successors;
      ())
    g;
  !count
```

Examples

- Iteration
- Currying

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

Examples

- Iteration
- Currying

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

is not:

```
val iter : local_ ('a -> unit) -> ('a list -> unit)
```

Examples

- Iteration
- Currying

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

is not:

```
val iter : local_ ('a -> unit) -> ('a list -> unit)
```

but instead:

```
val iter : local_ ('a -> unit) -> local_ ('a list -> unit)
```

Examples

- Iteration
- Currying

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

is not:

```
val iter : local_ ('a -> unit) -> ('a list -> unit)
```

but instead:

```
val iter : local_ ('a -> unit) -> local_ ('a list -> unit)
```

```
let f = List.iter g
```

Examples

- Iteration
- Currying
- Local functions

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

Examples

- Iteration

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

- Currying
- Local functions

```
val with_file :
  filename:string ->
  local_ (local_ filehandle -> 'a) ->
  'a
```

Examples

- Iteration
- Currying
- Local functions

```
val iter : local_ ('a -> unit) -> 'a list -> unit
```

```
val with_file :
  filename:string ->
  local_ (local_ filehandle -> 'a) ->
  'a
```

```
val immut_array :
  length:int ->
  init:'a ->
  local_ (local_ 'a array -> 'b) ->
  'a immut_array * 'b
```

Examples

- Iteration
- Currying
- Local functions
- **More uses**

```
val borrow :
   unique_ 'a ->
   local_ (local_ 'a -> 'b) ->
   unique_ 'a * 'b
```

Examples

- Iteration
- Currying
- Local functions
- More uses

```
val borrow :
   unique_ 'a ->
   local_ (local_ 'a -> 'b) ->
   unique_ 'a * 'b
```

```
val effectful :
   local_ 'a handler -> unit
```

Conclusion

Stack allocation is **efficient**...

Conclusion

Stack allocation is **efficient**...

... but locals are useful for **more than speed**.

Conclusion

Stack allocation is **efficient**...

... but locals are useful for **more than speed**.

Code & docs at:

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

`{sdolan,lwhite}@janestreet.com`