What If We Don't Pop the Stack? The Return of 2nd-Class Values

Anxhelo Xhebraj
Purdue University, West Lafayette, USA

Oliver Bračevac
Purdue University, West Lafayette, USA

Guannan Wei
Purdue University, West Lafayette, USA

Tiark Rompf
Purdue University, West Lafayette, USA

Abstract

Using a stack for managing the local state of procedures as popularized by Algol is a simple but effective way to achieve a primitive form of automatic memory management. Hence, the call stack remains the backbone of most programming language runtimes to the present day. However, the appealing simplicity of the call stack model comes at the price of strictly enforced limitations: since every function return pops the stack, it is difficult to return stack-allocated data from a callee upwards to its caller—especially *variable-size* data such as closures.

This paper proposes a solution by introducing a small tweak to the usual stack semantics. We design a type system that tracks the underlying *storage mode* of values, and when a function returns a stack-allocated value, we *just don't pop the stack*! Instead, the stack frame is de-allocated together with a parent the next time a heap-allocated value or primitive is returned. We identify a range of use cases where this delayed-popping strategy is beneficial, ranging from closures to trait objects to other types of variable-size data. Our evaluation shows that this execution model reduces heap and GC pressure and recovers spatial locality of programs improving execution time between 10% and 25% with respect to standard execution.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Call stack, closures, stack allocation, memory management, 2nd-class values, capabilities, effects

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.15

1 Introduction

Using a call stack to manage activation records of procedures was one of the great advances in the design and implementation of programming languages. Discovered by Bauer in the 1950s [11] and popularized by Dijkstra and others in the design of Algol in the 1960s [23, 29], the call stack enabled general recursion by supporting multiple concurrent activations of the same function [23], a significant gain in expressiveness over early Fortran dialects and other languages of the time. The call stack also provides a simple but effective form of automatic memory management, which makes it the backbone of almost every programming language runtime to the present day (with some notable exceptions, e.g., SML/NJ [4, 37]).

However, the appealing simplicity of the call stack model comes at the price of strictly enforced limitations. Since *every* function return pops the stack, it is difficult to return stack-allocated data from a callee upwards to its caller—especially variable-size data such as closures. Existing language implementations sidestep this issue by allocating return values beyond a small fixed size on the heap (forgoing benefits of stack allocation such as guaranteed timely deallocation), by trying to infer or bound the size of returned values and reserving

sufficient space in the parent stack frame (not always possible and sometimes wasteful), or by abandoning the rigid call stack altogether in favor of more flexible structures such as stacks of resizable regions (with overall increased complexity and loss of performance predictability).

This work is based on the observation that in cases where returning a stack-allocated value is desired, the value's lifetime is typically still bounded, it just needs to live *a little bit* longer than the procedure that allocated it. So, what would happen if we *just don't pop the stack* and delay it until one of the callers resets the stack, popping multiple frames at once? It turns out that this surprisingly simple idea can be made to work rather well. When returning a stack-allocated value, we retain the callee stack frame, essentially treating it as a block allocated as part of the parent stack frame. The stack is reset once a primitive or heap-allocated value is returned, resulting in joint deallocation of multiple stack frames. We present a type system that tracks the underlying storage type of references to guide reclamation decisions. The key safety invariant is that heap-allocated data does not contain stack references, and that stack references only point upward, not downward.

The central appeal of this "delayed popping" strategy is its simplicity and attractive power-to-weight ratio. Since the approach constitutes only a minor adjustment to the ubiquituous call stack model, it should be easy to add to almost any language implementation supporting stack allocation. All the necessary safety checks can be retrofitted onto existing type systems that track scoped lifetimes of values. In particular, we extend Osvald et al.'s [45] $\lambda^{1/2}$ type system for tracking 2nd-class values, achieving considerable gains in expressiveness (e.g., curried 2nd-class functions [16]). Delayed popping and returning variable-size data on the stack benefit a variety of uses cases, ranging from closures to trait objects to other types of data. Reducing heap and GC pressure and increasing spatial locality significantly improves end-to-end execution time. Certain classes of programs previously dominated by GC overhead can now run entirely on the stack.

In summary, this paper makes the following contributions:

- We discuss the problem of returning stack-allocated values and describe our solution of delaying stack frame reclamation informally using examples (Section 2).
- We present the $\lambda^{1/2}_{\hookleftarrow}$ type system for tracking *storage-mode* qualifiers. To ensure validity of references in the presence of both heap and stack references, we attach simple qualifiers to types [25] and restrict stack-allocated values to a *2nd-class* status (Section 3).
- We present the operational semantics of our model, and we establish key properties: (1) type safety: well-typed programs do not go wrong, (2) memory separation: 1st-class values do not refer to 2nd-class values, (3) stack allocation: 2nd-class values can be stack-allocated with delayed popping (Section 4). We have mechanized the proofs in Coq.
- We present compiler implementations of our approach (1) in Scala Native [50] with an LLVM backend using a shadow stack, and (2) in MiniScala with an x86-64 backend using the native call stack (Section 5).
- We provide in-depth discussions of extensions and uses cases, e.g., support for parametric polymorphism (including abstracting over storage modes), on-stack mutable data, programming with capabilities, and how we overcome limitations of previous work on 2nd-class values and existing language implementations (e.g., Rust) when returning variable-size data on the stack (Section 6).
- We evaluate our system on a variety of benchmarks showing speedups between 10% and 25% and present two in-depth case studies on static memory management for deep learning workloads and parser combinators (Section 7).

We discuss related work in Section 8 and conclude in Section 9. Our artifacts (implementation, Coq proofs) are available online at `https://github.com/angelogeb/scala-native`.

2 The Return of Stack-Allocated Values

Since the days of Algol [11, 23, 29], stack-allocated activation records (*stack frames*) are the core data structure for implementing multiple activations of the same function. Stack frames are instantiated when performing a function call and used for: (1) storing the return address of the calling procedure, (2) passing the parameters to the called procedure, (3) storing temporary local variables that are de-allocated at the end of the procedure, and (4) passing returned values back to the caller. In this paper we are interested in (3), the stack allocation of values, and (4), passing values back to the calling procedure on the stack.

The goal: returning stack-allocated data. While dealing with stack-allocated primitives of a fixed size (such as 4 or 8 bytes) is trivial, returning stack-allocated compound data types raises some issues. Consider computing the norm of the sum of two vectors $\|v_1 + v_2\|_2$ using a library that provides the functions `add` and `norm`:

```
norm(add(v1, v2))
```

The function `add` needs to return an array denoting the sum of `v1` and `v2`. However, returning an array typically requires allocating its storage on the heap, unless one is willing to copy a potentially sizable chunk of memory repeatedly between stack positions. Heap allocation is not ideal in this case either because the lifetime of the value is known: it is immediately consumed by `norm`. Moreover, the allocation will have to be reclaimed in ways that are unsafe (manually), have a performance penalty (garbage collection or reference counting) or require more complicated type systems based on lifetimes and borrowing.

Destination-passing style: a manual workaround. To work around this issue, code bases written in Fortran or C from the domain of High Performance Computing (HPC) often design functions such as `add` in *Destination-Passing Style* (DPS) [33, 51]. In this setting, every function that needs to return an array accepts a pre-allocated result buffer as argument, so that the caller is in control of the result's memory management:

```
@stack val vout = new Vec[f32](v1.length)
add(v1, v2, vout)
norm(vout)
```

We use Scala Native in this paper, but the discussion equally applies to any language that supports stack allocation. We use the `@stack val ...` annotation to denote stack-allocated values, building on a recently proposed type system by Osvald et al. [45] to guarantee that such "2nd-class" values do not escape. In C, the example could be equivalently written using a local array declaration or explicitly using the stack allocation primitive `alloca(v1.length * sizeof(float))`, although without any safety checking. We describe our model, including the type system, in more detail in Section 2.2. The DPS implementation of `norm` and `add` is shown in Figure 1a.

In the example above, the caller decides to allocate the result buffer on the stack, which means that the buffer will be de-allocated automatically at the end of the scope. However, this comes with downsides too — we lose the ability to write expression-oriented code like `norm(add(v1, v2))` or perhaps `norm(v1 + v2)` with proper operator overloading, and potential sharing of buffers can lead to subtle bugs. For example, a call like `movingAverage(vin=v, vout=v)` would produce an incorrect value, since the output at index `i` depends on the input at index `i-1`, which would have been overwritten by the previous iteration:

```
def movingAverage(vin: Vec[f32] @stack, vout: Vec[f32] @stack) = {
  for (i <- 1 until (vin.length - 1))
    vout(i) = (vin(i - 1) + vin(i) + vin(i + 1)) / 3
}
```

Variable-size data: the key limitation of destination passing. In the case of `add`, it is possible for the caller to provision memory for the callee and allocate the result buffer on the stack, but there are cases where this is not possible and we have to rely on the heap instead. Consider a program that has to deserialize data using a utility function `readNextGroup` which reads a data-dependent number of bytes from a stream:

```
def readNextGroup(f: FStream) = {
  val len = readInt(f)   // data-dependent
  val res = new Vec[Byte](len) ① // alloc
  readBytes(res, len)
  res
}
```

```
def deserialize(f: FStream): Tree = {
  val buf = readNextGroup(f) // unknown result size:
  buf(0) match {             // can't pre-allocate!
    case I32_TAG => atoi(buf)
    ...
} }
```

Unlike the `add` example, the result's size is data dependent and therefore it is not possible to manage memory in DPS, unless we were to break modularity and function boundaries by inlining. Instead, the code must allocate the result buffer on the heap (①). Both examples need to work around the inability to *return short-lived dynamically sized* objects on the stack.

Closures: a particular important case of variable-size data. The sizing restriction of the strict stack discipline also conflicts with higher-order functions. Programming languages based on stack environments rely on types to compute a value's storage size. However, a closure's type does not describe its closing environment and, unlike the array examples, a solution attempt would neccessarily involve some form of defunctionalization [22] and intensional type analysis, again breaking modularity.

```
def useCurried(f: Int => Int => Int) = {
  val g = f(10) // unknown result size:
  ...           // can't pre-allocate!
}
```

```
val (x, y) = ...
useCurried(_ => (_ => 1)①)
useCurried(a => (b => a * x + b * y)②)
```

In the program above, the size of the closure `g` returned by `f` is statically unknown. In the first call, the returned closure (①) has a trivial size. In the second call, the returned closure (②) captures `x` and `y` and therefore has a larger size than (①). Pre-allocating memory for `g` would require knowing all possible closures that can be returned and reserving space required for the largest one. In most cases this is infeasible, therefore returning closures requires forgoing stack allocations and relying extensively on the heap (see also Section 6.7).

Problem summary: always popping the stack is inflexible. The root of the problem is that every function is required to pop its stack frame immediately when returning. Under a *strict stack discipline* it is possible to return a value on the stack *only if its size can be statically upper-bounded at the call site*. In such cases, storage for the returned value is reserved on the caller's stack frame, a reference to the storage is passed to the callee and a copy of the returned value is performed by the callee (DPS). This is necessary to ensure that a function return resets the top of the stack to what it was before the function call.

2.1 A Partial Solution: 2nd-class Values and Selective CPS Conversion

Before looking at ways to relax the strict invariant that every function must pop its stack frame immediately when returning, we consider strategies that inspired our solution.

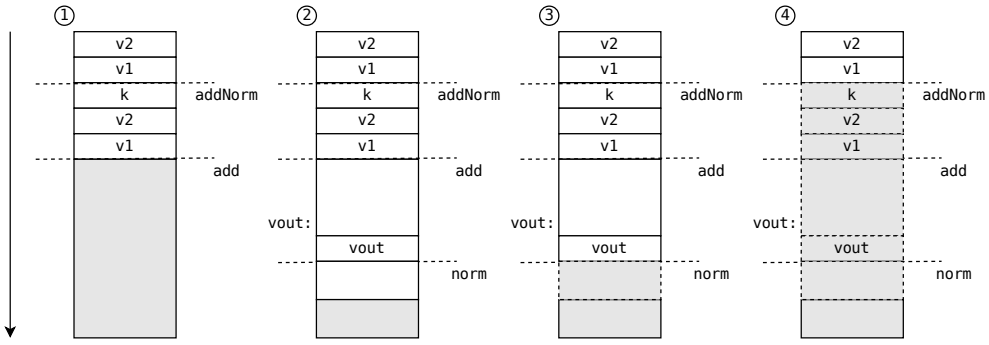
Selective CPS conversion to extend stack lifetime. In the presence of higher-order functions, we can eschew the problem of returning by transforming programs to Continuation-Passing Style (CPS) [47]. Rather than passing a destination address to the caller as in DPS, we pass a callback (the continuation) to receive the result address. The CPS transformation enables rewriting programs that return function values into equivalent programs where functions appear only in argument positions or as operands in call expressions (in the style of Algol) [24], including all cases that return *compound data types of unknown size* [38].

```
def norm(v: Vec[f32] @stack): f32 = ...
def add(v1 : Vec[f32] @stack, v2: Vec[f32] @stack, res: Vec[f32] @stack): Unit = {
  for (i <- 0 until v1.length)
    res(i) = v1(i) + v2(i)
}
def addNorm(v1: Vec[f32] @stack, v2: Vec[f32] @stack): f32 = {
  @stack val vout = new Vec[f32](v1.length)
  add(v1, v2, vout)
  norm(vout)
}
```

(a) Destination passing style: having the caller preallocate storage is a standard workaround for "returning" stack-allocated values. This approach only works if the caller knows the (maximum) size of the result.

```
def add[T](v1: Vec[f32] @stack, v2: Vec[f32] @stack, k: (Vec[f32] @stack => T) @stack): T = { ①
  @stack val vout = new Vec[f32](v1.length)
  for (i <- v1.length)
    vout(i) = v1(i) + v2(i)
  k(vout) ②
}
def addNorm(v1: Vec[f32] @stack, v2: Vec[f32] @stack): f32 = {
  add(v1, v2, norm ③)
} ④
```

(b) "Returning" a value on the stack through a selective CPS transform. The stack is managed as depicted in (c). Before the call to `add`, `addNorm` pushes the arguments on the stack (①). In ② `add` allocates the array on the stack and passes its reference to `norm`. After `norm`'s return (③), both `add`'s and `addNorm`'s stack frame will be reclaimed (④).



(c) Stack behavior of selective CPS version (b) when `addNorm` is called with references to the argument arrays `v1`, `v2` on the stack: right after entering `add`'s body (①), after calling the continuation `norm` (②), after `norm`'s return (③) and after `addNorm`'s return (④). Grey portions of the stack are free.

```
def add(v1: Vec[f32] @stack, v2: Vec[f32] @stack): Vec[f32] @stack = { ①
  @stack val vout = new Vec[f32](v1.length)
  for (i <- v1.length)
    vout(i) = v1(i) + v2(i)
  vout
}
def addNorm(v1: Vec[f32] @stack, v2: Vec[f32] @stack): f32 = {
  norm(add(v1, v2)②)③
} ④
```

(d) Returning a value on the stack through storage modes. After ①, `add`'s stack frame is retained to later be popped when `addNorm` returns (④). The stack behavior is the one depicted in (c) without `k`.

.

Figure 1 Alternatives for stack-allocated arrays: DPS (a), selective CPS (b) and its stack behavior (c), direct style with storage modes (d). Storage modes emulate the selective CPS stack behavior.

We can rewrite function `add` in CPS as shown in Figure 1b. Operationally the CPS transformation delays the reclamation of `add`'s activation record to the point where the continuation `k` returns. Instead of returning a value on the stack, `add` accepts a continuation that can use the "returned" value in non-escaping fashion. Figure 1c illustrates the stack behavior after the CPS rewrite. Before the call to `add`, `run` pushes the arguments on the stack (①). In ②, `add` has allocated the array on the stack and passes its reference to `norm`. After `norm` returns, both `add`'s and `run`'s stack frame will be reclaimed (③).

This solution achieves our initial goal, i.e., `add` can now "return" variable-size data. Importantly, the CPS transformation should be selective [40, 49, 7] since transforming all definitions into CPS would eliminate the call stack altogether [8, 38]. However, the solution is not ideal because it requires (1) selective non-local transformations of the code, and (2) it might result in even more heap allocations if continuations are not dealt with properly. Nevertheless, the CPS version of the program shows what is required to return variable-size stack-allocated values – deferring the de-allocation of the stack frame of the callee and a type system that ensures validity of stack references.

Tracking non-escaping values in types. The $\lambda^{1/2}$ type system by Osvald et al. [45] allows to implement this pattern safely. Their system qualifies values as "2nd class" through the `@local` annotation on types (we use `@stack` to denote the same in this paper). Such 2nd-class values are not allowed to escape their defining scope, i.e., their lifetimes follow a strict stack discipline. The $\lambda^{1/2}$ type system enforces that:

1. 1st-class functions may not refer to 2nd-class values through free variables.
2. Functions may not return 2nd-class values.
3. 2nd-class values may not be stored in mutable variables or object fields.

This ensures that 2nd-class references are used in a non-escaping fashion. The key safety invariant is that heap-allocated data is "1st class" and does not contain stack references, and that stack references only point upward, not downward. CPS preserves the precise stack behavior of 2nd-class values [19]. Crucially, $\lambda^{1/2}$ *disallows* returning 2nd-class values to maintain strict stack safety. A key contribution of this paper is to lift this restriction in a sound way, leading to a model of delayed reclamation of stack frames that exhibits the same desired behavior as the selective CPS conversion, but in *direct style*.

2.2 Our Solution: Delay Popping in Direct Style, Using Type Qualifiers

Based on the preceding insights, we propose a model to allow returning short-lived stack-allocated values for which sizes are unknown at compile time for programs *in direct style*, by allowing functions to return references to their stack frame and using Osvald-style [45] storage-mode qualifiers to guide stack reclamation.

Figure 1d shows the implementation of `addNorm` in direct style using storage modes. When `add` returns `vout`, its stack frame is retained, relaxing the *too strict* stack discipline. To this end, we enrich Osvald et al.'s type system [45] with `@stack` qualifiers *on function return types*. These qualifiers drive the operational behavior in crucial ways:

1. When returning a 2nd-class value, do not pop the stack.
2. In that situation, defer the deallocation of the callee's stack frame to a point where a caller further up the stack returns a 1st class value.
3. When returning a 1st-class value, reset the stack as usual; this will reclaim all stack frames allocated by callees.

This strategy is safe because 1st-class values cannot refer to 2nd-class ones, and 2nd-class values do not escape other than through the return path, emulating the CPS stack behavior (Figure 1). We break with the convention that the top of the stack remains unchanged after

function applications returning `@stack`-qualified values. It is important to note that this does not interfere in major ways with the caller's stack layout. As long as the caller maintains a pointer to the start of its own stack frame, it can treat the remaining callee stack frame like any other piece of stack-allocated data and continue allocating at the current stack pointer, pointing to the end of the stack frame (see also Section 5).

Controlling allocations through storage-mode qualifiers. We provide further examples to showcase our programming model. We assume a call-by-value language with primitive types, compound types (e.g., closures and arrays), and automatic memory management, representative of languages like Java, OCaml, or Scala. Values are either constants of primitive types or references to values of compound types (either allocated on the stack or a heap). Closures capture the smallest environment by value and store it in the closure object.

Stack bindings and storage modes. Values can be allocated on the stack through `@stack` bindings. In the following example `inc1` is a stack-allocated closure:

```
def add1(l: List[Int]) = {      // : List[Int] => List[Int]
  @stack val inc1 = i => i + 1  // : (Int => Int) @stack
  map(l, inc1)                  // : List[Int]
}
add1(List(1,2,3,4))             // = List(2,3,4,5)
```

Stack bindings induce a `@stack` annotation *on the type* of the bound variable, called *storage-mode qualifiers*. Stack bindings can be omitted if they can be inferred from types, e.g., we can equivalently annotate `inc1` with a type ascription:

```
val inc1: (Int => Int) @stack = i => i + 1
```

where the `@stack` qualifier is attached to the function type. We can also rewrite the body of `add1` more concisely as

```
map(l, i => i + 1)              // stack allocation inferred
```

and let type inference assign the desired storage mode to the function argument of `map`, as explained next.

Storage-mode statics. While unannotated values (1st-class values) can be used freely, the type system ensures that `@stack`-qualified values do not escape into 1st class contexts, i.e., non-`@stack`-qualified positions. For the example above, the type of `map`'s second argument *must* be `@stack` qualified as shown below (①):

```
def map[I, O](l: List[I], f: (I => O) @stack①): List[O] = l match {
  case Nil => Nil
  case Cons(h, t) => Cons(f(h), t)
}
```

However, the `@stack` qualifier only restricts the uses of `f` but does not mandatorily induce a stack allocation, i.e., we can still pass heap-allocated functions to `map` (see also Section 6.2):

```
val f = (v: Int) => v + 10      // closure allocated on the heap
map(l, f)                       // ok
```

This is achieved by the subtyping relationship `@heap <: @stack` where unqualified types `T` have an implicit qualifier `@heap` (1st class). Storage modes and their static semantics ensure that *heap-allocated objects do not refer to stack-allocated ones.*

Storage-mode dynamics. Storage modes permit returning variable-size data on the stack by retaining the callee's stack frame when it returns a `@stack`-qualified value. The stack frame is popped once reaching a caller returning a `@heap`-qualified value. We support returning any type of variable-size data, even recursive ones such as lists (Figure 2), the `filter` function builds the result list on the stack. After the call to `filter`, the execution returns to `run` with the updated stack pointer (②). The returned list is passed as an argument to `sum`, which will

```
def filter(l: List[Int] @stack, f: (Int => Boolean) @stack): List[Int] @stack = l match {
  case Nil => Nil
  case Cons(h, t) =>
    val t' = filter(t, f)
    if (f(h)) Cons(h, t') else t'
}
def run(l: List[Int]): Boolean = { ①
   val l' = filter(l, x => x %
② sum(l') > 0
}
val l = List(1,2,3,4,5)
run(l)
```

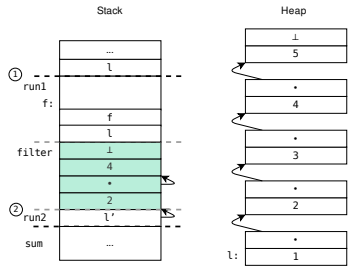


Figure 2 Example program that returns a list on the stack. `filter`'s stack frame is retained after its return (shown in cyan). Note that `filter` also works with heap-allocated list arguments.

execute its body and pop its stack frame. Once `run` returns, it will pop its stack frame, too (since it returns a `@heap`-qualified value[1]), which also contains `filter`'s stack frame.

Values with a `@stack` qualifier are 2nd class. As noted above, storage modes are an instance of Osvald et al.'s 2nd-class values [45]. Their work proposed to reintroduce 2nd-class values in the style of Algol's procedure parameters to implement capabilities for safe exceptions and, more generally, effect checking. To achieve this, their type system accepts only function definitions that have 1st-class return types to ensure that capabilities conform to the strict stack discipline. However, this is restrictive and inhibits many use cases where a returned value has to close over a 2nd-class argument, such as currying over a capability.

Our work removes this limitation and employs qualifiers, not only to ensure the validity of references but also to change the runtime behavior of the stack.

3 The $\lambda^{1/2}_{\hookleftarrow}$ Storage-Mode Qualifier Calculus

We formalize storage modes in the $\lambda^{1/2}_{\hookleftarrow}$-calculus, an extension of the $\lambda^{1/2}$-calculus by Osvald et al. [45], which distinguishes between 1st-class and 2nd-class values.[2] In the $\lambda^{1/2}$ system, functions cannot return 2nd-class values, implying strict stack-based 2nd-class lifetimes. Our $\lambda^{1/2}_{\hookleftarrow}$-calculus eliminates this restriction, yielding considerable gains in expressiveness, e.g, it supports currying of functions that take 2nd-class arguments, whereas $\lambda^{1/2}$ does not. We prove type soundness for $\lambda^{1/2}_{\hookleftarrow}$ with respect to the standard call-by-value semantics of the λ-calculus. This lays the groundwork for proving stronger results about memory invariants, such as safety and correctness of stack allocation with delayed popping (Section 4).[3]

3.1 Syntax and Typing Rules

Terms in $\lambda^{1/2}_{\hookleftarrow}$ (Figure 3) can be constant literals of base types, variables, functions, and applications. We annotate types with storage-mode qualifiers q. These can be "1" denoting a 1st-class/heap-allocated result, or "2" denoting a 2nd-class/stack-allocated result. Storage modes are totally ordered, where $1 \leq 2$.

Typing environments Γ (Figure 3) come with the usual *lookup* operator $\Gamma(x) = T^q$ and a *filter* operator $\Gamma^{[\leq q]}$ that returns all the assumptions $x : T^{q_x}$ in Γ satisfying $q_x \leq q$.

---

[1] Scalar primitives are always 1st class (`@heap` qualified) and therefore can be used freely.
[2] Our system also scales to $F_{<:}$-style parametric polymorphism using path-dependent types (cf. [61]).
[3] We mechanized these result in Coq, available at `https://github.com/angelogeb/scala-native`.

Syntax $\lambda^{1/2}_{\hookleftarrow}$

$$q ::= 1 \mid 2 \qquad t ::= c \mid x \mid \lambda x.t \mid t\ t \qquad T ::= B \mid T^q \to T^{\,q} \qquad \Gamma ::= \varnothing \mid \Gamma, x : T^q$$

$$\Gamma^{[\leq q]} := \{x : T^{q_x} \in \Gamma \mid q_x \leq q\}$$

Subtyping Rules $T^{q_1} <: T^{q_2}$

$$\frac{q_1 \leq q_2}{B^{q_1} <: B^{q_2}}\ \text{SRfl} \qquad \frac{T_3^{q_3} <: T_1^{q_1} \qquad T_2^{\,q_2} <: T_4^{\,q_4} \qquad q_5 \leq q_6}{(T_1^{q_1} \to T_2^{\,q_2})^{q_5} <: (T_3^{q_3} \to T_4^{\,q_4})^{q_6}}\ \text{SFun}$$

Typing Rules $\Gamma \vdash t : T^q$

$$\frac{}{\Gamma \vdash c : B^q}\ \text{TCst} \qquad \frac{\Gamma(x) = T^q}{\Gamma \vdash x : T^q}\ \text{TVar} \qquad \frac{\Gamma^{[\leq q]}, x : T_1^{q_1} \vdash t : T_2^{\,q_2}}{\Gamma \vdash \lambda x.t : (T_1^{q_1} \to T_2^{\,q_2})^{\,q}}\ \text{TAbs}$$

$$\frac{\Gamma \vdash t_1 : (T_1^{q_1} \to T_2^{\,q_2})^{\,2} \qquad \Gamma \vdash t_2 : T_1^{q_1}}{\Gamma \vdash t_1\, t_2 : T_2^{\,q_2}}\ \text{TApp} \qquad \frac{\Gamma \vdash t : T_1^{q_1} \qquad T_1^{q_1} <: T_2^{q_2}}{\Gamma \vdash t : T_2^{q_2}}\ \text{TSub}$$

Figure 3 The $\lambda^{1/2}_{\hookleftarrow}$ type system. Differences to Osvald et al.'s system [45] are highlighted.

The type system (Figure 3) is an extension of the simply-typed λ-calculus (STLC) with subtyping plus storage-mode qualifiers on types. Qualifiers propagate through the subtyping rules, essentially allowing the use of 1st-class terms in positions where 2nd-class terms are expected (SRfl), and they are subject to the usual contravariance in function domains and covariance in function codomains (SFun). Filtering the typing context when typing function bodies (TAbs) ensures that 1st-class functions do not capture 2nd-class values.[4]

The subtle yet crucial difference to Osvald et al.'s $\lambda^{1/2}$-calculus is that $\lambda^{1/2}_{\hookleftarrow}$ also attaches qualifiers to *codomains* of function types, granting more degrees of freedom when typing functions (TAbs) and applications (TApp); e.g., this enables currying of 2nd-class parameters.

3.2 Type Soundness of Standard Small-Step Evaluation

We use the standard call-by-value (cbv) single-step reduction of the λ-calculus, and prove type soundness via progress and preservation lemmas [60]. The proofs require reflexivity and transitivity of subtyping, as well as weakening and narrowing lemmas, which are standard.

▶ **Theorem 1** (Progress). *Given a closed term t that is well-typed $\varnothing \vdash t : T^q$ then either t is a value or else there exists t' such that $t \longrightarrow t'$.*

Proof. By induction on the derivation $\varnothing \vdash t : T^q$, using canonical forms lemmas. ◀

▶ **Theorem 2** (Preservation). *Given a closed term t that is well-typed with type T^q, i.e., $\varnothing \vdash t : T^q$, if t steps to t', $t \longrightarrow t'$, then $\varnothing \vdash t' : T^q$.*

Proof. By induction on the derivation $\varnothing \vdash t : T^q$, using the usual lemmas for inversion of typing and preservation under substitution. ◀

From progress and preservation, we can establish type soundness of the evaluation semantics induced by the single-step reduction relation [60].

[4] For simplicity, the base system lacks recursive functions. These can be readily added by allowing self-references in lambdas ($\lambda f(x).t$), adding $f : (T_1^{q_1} \to T_2^{q_2})^q$ to the body's context in TAbs, and generalizing TApp to make use of q.

Big-Step Evaluation: Standard and Instrumented $\boxed{\mathcal{H} \vdash t \Downarrow^q v}$

$$t ::= c \mid x^q \mid \lambda x^q.t^q \mid t\ t \qquad v ::= c \mid \langle \mathcal{H}, \lambda x^{q_1}.t^{q_2} \rangle \qquad \mathcal{H} ::= \varnothing \mid \mathcal{H}, x^q : v$$

$$\mathcal{H}^{[\leq q]} := \{(x^{q_x} : v) \in \mathcal{H} \mid q_x \leq q\}$$

$$\frac{}{\mathcal{H} \vdash c \Downarrow^q c}\ \text{QECst} \qquad \frac{\mathcal{H}^{[\leq q]}(x^{q_1}) = v}{\mathcal{H} \vdash x^{q_1} \Downarrow^q v}\ \text{QEVar} \qquad \frac{}{\mathcal{H} \vdash \lambda x^{q_1}.t^{q_2} \Downarrow^q \langle \mathcal{H}^{[\leq q]}, \lambda x^{q_1}.t^{q_2} \rangle}\ \text{QEAbs}$$

$$\frac{\mathcal{H} \vdash t_1 \Downarrow^2 \langle \mathcal{H}', \lambda x^{q_2}.t_3^{q_3} \rangle \qquad \mathcal{H} \vdash t_2 \Downarrow^{q_2} v_2 \qquad \mathcal{H}', x^{q_2} : v_2 \vdash t_3 \Downarrow^{q_3} v_3 \qquad q_3 \leq q}{\mathcal{H} \vdash t_1\ t_2 \Downarrow^q v_3}\ \text{QEApp}$$

Figure 4 Two big-step semantics for $\lambda^{1/2}_{\hookleftarrow}$ using environments. (1) Excluding **teal** parts: the standard call-by-value big-step semantics of the λ-calculus. (2) Including **teal** parts: a more restrictive semantics that internalizes storage modes in the term syntax and checks storage modes.

4 Memory Properties

We have established in Section 3.2 that "well-typed $\lambda^{1/2}_{\hookleftarrow}$ programs do not go wrong" in terms of the cbv λ-calculus reduction semantics. While already important, this only asserts that well-typed $\lambda^{1/2}_{\hookleftarrow}$ terms do not exhibit the runtime errors of the ordinary λ-calculus. However, we need to prove further memory properties: (1) 1st-class values never close over 2nd-class values, and (2) delayed popping of the stack is safe. The solution is *refining* the semantics to check for *more* runtime errors, and prove that type soundness still holds.

4.1 1st-Class Values Never Capture 2nd-Class Values

As a first refinement, we let the term syntax carry explicit qualifier annotations q (obtainable from typing derivations) and define a new "instrumented" evaluation semantics checking for class violations at runtime, resulting in the calculus $\lambda^{1/2}_{q\hookleftarrow}$ (Figure 4, semantics 2). It augments the standard big-step semantics of the cbv λ-calculus with the colored parts.

We check for qualifier mismatches in lookup (rule QEVar) and that the qualifier of a closure's function body agrees with the one under evaluation (rule QEApp). When building closures in rule QEAbs, only a subset of the environment is captured, which is enforced by filtering the environment with the current class context q. These changes (1) implicitly partition $\mathcal{H}$ into a 1st- and 2nd-class environment, and (2) make evaluation stuck if a 1st-class/heap-allocated value captures a 2nd-class/stack-allocated value.

Proof technique. We model Wright and Felleisen's "strong soundness" notion [60] using a total evaluator function[5] in the style of Siek [52] and Amin and Rompf [3]

$$\mathsf{eval}^q : \mathbb{N} \to \mathcal{H} \to t \to (\mathsf{Done}\ (\mathsf{Val}\ v \mid \mathsf{Wrong}) \mid \mathsf{Timeout})$$

which, given a fuel value $k \in \mathbb{N}$ and a runtime environment $\mathcal{H}$, evaluates a term to a result r that can be either (1) Timeout if the fuel is not enough to complete the evaluation, or (2) Done Wrong in case of a runtime error, or (3) Done (Val v) for a result value v. Here, eval^q implements the big-step evaluation relation $\mathcal{H} \vdash t \Downarrow^q v$ (Figure 4), i.e., $\mathcal{H} \vdash t \Downarrow^q v$ if and only if there is a fuel value $k \in \mathbb{N}$ such that $\mathsf{eval}^q\ k\ \mathcal{H}\ t = \mathsf{Done}\ (\mathsf{Val}\ v)$.

[5] This proof style is more succinct for proving the sought-after runtime invariants, because it models closures explicitly. The switch to big step is justifiable, because small- vs. big-step semantics, and substitution- vs. environment-based semantics are known to be equivalent [13, 1, 20].

Type soundness implies memory separation. The strong soundness proof depends on well-typed values ($v : T^q$) and well-formed environments ($\Gamma \vDash \mathcal{H}$), defined below. The key property is demanding that well-typed closures capture *only* values below or at their assigned class, *and nothing else*, as follows:

$$\frac{}{c : B^q} \qquad \frac{\Gamma \vDash \mathcal{H} \quad \mathcal{H}^{[\leq q]} = \mathcal{H} \quad \Gamma, x : T_1^{q_1} \vdash t : T_2^{q_2}}{\langle \mathcal{H}, \lambda x^{q_1}.t^{q_2} \rangle : (T_1^{q_1} \to T_2^{q_2})^q} \qquad \frac{}{\varnothing \vDash \varnothing} \qquad \frac{\Gamma \vDash \mathcal{H} \quad v : T^q}{\Gamma, x : T^q \vDash \mathcal{H}, x^q : v}$$

▶ **Theorem 3** (Strong Soundness). *The $\lambda^{1/2}_{\hookleftarrow}$ type system is sound with respect to the instrumented big-step semantics (Figure 4, semantics 2): For all q, and for all k, if* eval^q *does not time out, then its result is a well-typed value:*

$$\frac{\Gamma \vdash t : T^q \qquad \Gamma \vDash \mathcal{H} \qquad \mathsf{eval}^q\ k\ \mathcal{H}\ t' = \mathsf{Done}\ r}{r = \mathsf{Val}\ v \qquad v : T^q}$$

Proof. By induction on the fuel value k, and case analysis on the term t, using helper lemmas to establish soundness of environment lookup. ◀

Intuitively, due to value typing and the extra class violation checks in the instrumented semantics, the strong soundness Theorem 3 implies:

▶ **Corollary 4.** *Well-typed 1st-class functions never capture 2nd-class values.*

Proof. By Theorem 3 and the definition of well-typed values for 1st-class function types. ◀

4.2 Stack-based Evaluation with Deferred Popping is Safe

As a further refinement, we design a semantics where 2nd-class bindings follow a delayed stack discipline and thus permit a corresponding practical call-stack implementation. Figure 5 shows the evaluation rules of the refined semantics with stacks. The big-step relation $\mathsf{H}, \mathsf{S} \vdash t \Downarrow^q_{\mathsf{s}} v \dashv \mathsf{S}'$ accepts as input an environment H, a stack S, a term t, and qualifier q, producing an output value v and a new stack S'. The stack is effectively a piece of state, threaded through computations. An environment H is an association list as usual while a stack S is a list of frames, where a frame Φ is also an association list of bindings. Stacks are *snoc* lists, with the head element having the largest index. Occasionally, we use the notation $\Phi_0 \ldots \Phi_k$ to visualize the stack and the corresponding index of each frame. The environment lookup $(\mathsf{H}, \Phi)(x^q)$ depends on the variable's qualifier q (rules EVarH and EVarS), e.g., we look up 2nd-class variables in the topmost stack frame. The same applies for environment extension $(\mathsf{H}, \Phi) \oplus x^q : v$ (cf. [61] for their formal definitions).

In contrast to the instrumented semantics (Figure 4), closures now contain a pointer *ptr* into the stack, which can be either $\bot$ (1st-class closures) or a natural number for the k-th stack frame (2nd-class closures). The *lookup*(S, ptr) operator (EAppH, EAppS) retrieves the k-th stack frame in S, if $ptr = k \in \mathbb{N}$, and otherwise a fresh stack frame if $ptr = \bot$.

Most rules are similar to their counterparts in Figure 4 and only read the input stack without updating it. When evaluating a 2nd-class function (EAbsS), its closure records the pointer $|\mathsf{S}|$ to the topmost frame. In contrast, a 1st-class function is not supposed to close over 2nd-class values and thus does not retain a pointer ($\bot$ in EAbsH). Both rules for 1st- and 2nd-class application (EAppH and EAppS) look up the closure's stack frame given its pointer and push it on top of the current stack to evaluate the function body. 1st-class application (EAppH) pops the stack after evaluating the body, i.e., the output stack equals the input stack S. In contrast, 2nd-class application (EAppH) *just does not pop the stack*,

Stack-based Big-Step Evaluation $\boxed{\mathsf{H}, \mathsf{S} \vdash t \;\Downarrow_{\mathsf{s}}^{q}\; v \dashv \mathsf{S}}$

$$\begin{array}{rrcl} \text{Pointer} & ptr & ::= & k \mid \bot \\ \text{Frames/Env.} & \Phi, \mathsf{H} & ::= & \mathsf{L} \\ \text{Value} & v & ::= & c \mid \langle \mathsf{H}, ptr, \lambda x^{q_1}.t^{q_2} \rangle \end{array} \qquad \begin{array}{rrcl} \text{List} & \mathsf{L}_k & ::= & \varnothing_{-1} \mid (\mathsf{L}_{k-1}, x^q : v)_k \\ \text{Stack} & \mathsf{S} & ::= & \varnothing \mid (\mathsf{S}, \Phi) \\ & k, i & \in & \mathbb{N} \end{array}$$

$$\textsc{ECst}\ \frac{}{\mathsf{H}, \mathsf{S} \vdash c \;\Downarrow_{\mathsf{s}}^{q}\; c \dashv \mathsf{S}} \qquad \textsc{EVarH}\ \frac{(\mathsf{H}, \varnothing)(x^1) = v}{\mathsf{H}, \mathsf{S} \vdash x^1 \;\Downarrow_{\mathsf{s}}^{1}\; v \dashv \mathsf{S}}$$

$$\textsc{EVarS}\ \frac{(\mathsf{H}, \Phi)(x^q) = v}{\mathsf{H}, (\mathsf{S}, \Phi) \vdash x^q \;\Downarrow_{\mathsf{s}}^{2}\; v \dashv (\mathsf{S}, \Phi)}$$

$$\textsc{EAbsH}\ \frac{}{\mathsf{H}, \mathsf{S} \vdash \lambda x^{q_1}.t^{q_2} \;\Downarrow_{\mathsf{s}}^{1}\; \langle \mathsf{H}, \bot, \lambda x^{q_1}.t^{q_2} \rangle \dashv \mathsf{S}} \qquad \textsc{EAbsS}\ \frac{}{\mathsf{H}, (\mathsf{S}, \Phi) \vdash \lambda x^{q_1}.t^{q_2} \;\Downarrow_{\mathsf{s}}^{2}\; \langle \mathsf{H}, |\mathsf{S}|, \lambda x^{q_1}.t^{q_2} \rangle \dashv (\mathsf{S}, \Phi)}$$

$$\textsc{EAppH}\ \frac{\begin{array}{c}\mathsf{H}, \mathsf{S} \vdash t_1 \;\Downarrow_{\mathsf{s}}^{2}\; \langle \mathsf{H}', ptr, \lambda x^{q_2}.t_3^{1} \rangle \dashv \mathsf{S}' \\ \Phi = lookup(\mathsf{S}', ptr) \\ \mathsf{H}, \mathsf{S}' \vdash t_2 \;\Downarrow_{\mathsf{s}}^{q_2}\; v_2 \dashv \mathsf{S}'' \\ (\mathsf{H}, (\mathsf{S}'', \Phi)) \oplus x^{q_2} : v_2 \vdash t_3 \;\Downarrow_{\mathsf{s}}^{1}\; v_3 \dashv \mathsf{S}'''\end{array}}{\mathsf{H}, \mathsf{S} \vdash t_1\, t_2 \;\Downarrow_{\mathsf{s}}^{1}\; v_3 \dashv \mathsf{S}}$$

$$\textsc{EAppS}\ \frac{\begin{array}{c}\mathsf{H}, \mathsf{S} \vdash t_1 \;\Downarrow_{\mathsf{s}}^{2}\; \langle \mathsf{H}', ptr, \lambda x^{q_2}.t_3^{q_3} \rangle \dashv \mathsf{S}' \\ \Phi = lookup(\mathsf{S}', ptr) \\ \mathsf{H}, \mathsf{S}' \vdash t_2 \;\Downarrow_{\mathsf{s}}^{q_2}\; v_2 \dashv \mathsf{S}'' \\ (\mathsf{H}, (\mathsf{S}'', \Phi)) \oplus x^{q_2} : v_2 \vdash t_3 \;\Downarrow_{\mathsf{s}}^{q_3}\; v_3 \dashv \mathsf{S}'''\end{array}}{\mathsf{H}, \mathsf{S} \vdash t_1\, t_2 \;\Downarrow_{\mathsf{s}}^{2}\; v_3 \dashv \mathsf{S}'''}$$

Figure 5 $\lambda_{q\hookleftarrow}^{1/2}$ big-step stack semantics. Important details regarding evaluation are highlighted: (1) the evaluation relation is classified with a qualifier; (2) closures retain a pointer indicating the stack they capture; (3) stack is also an "output" of the relation and is not popped in rule EAppS.

since the result v_3 might close over new 2nd-class bindings introduced during the body's evaluation. We define eval_s^q as the fuel-based interpreter corresponding to the stack-based evaluation relation.

Equivalence of stack and environment semantics implies safety. It is easy to recognize that the two semantics are equivalent in the sense that 2nd-class bindings are factored out of a common environment into an explicit stack and can only be captured through a pointer by closures. To this end, we define an equivalence relation $\mathsf{S} \vdash r_1 \sim r_2$ which relates the value, error, and divergence cases of the two semantics under a stack S. An environment $\mathcal{H}$ is equivalent to H, Φ if all of its values are equivalent to the values in the stack environment and vice versa (cf. [61] for the formal definition).

Equivalence is with respect to a stack because values in the stack semantics may capture 2nd-class references through stack pointers. Therefore, to relate them to values of the instrumented semantics, we must look up the right stack frame.

▶ **Theorem 5** (Equivalence of eval^q and eval_s^q)**.** *The instrumented environment semantics (Figure 4) and stack semantics (Figure 5) are equivalent, including timeout and error cases:*

$$\frac{(\mathsf{S}, \Phi) \vdash \mathcal{H} \sim \mathsf{H}, \Phi \qquad \mathsf{eval}^q\ k\ \mathcal{H}\ t\ = r_1 \qquad \mathsf{eval}_s^q\ k\ \mathsf{H}\ (\mathsf{S}, \Phi)\ t\ = (r_2,\ \mathsf{S}')}{\mathsf{S}' \vdash r_1 \sim r_2}$$

Proof. By induction on fuel value k and case analysis on term t. ◀

▶ **Corollary 6** (Strong Soundness)**.** *The $\lambda_{\hookleftarrow}^{1/2}$ type system is sound with respect to the stack-based evaluation semantics with delayed popping (Figure 5).*

Proof. By the soundness Theorem 3 for the instrumented semantics and the equivalence Theorem 5, well-typed $\lambda_{\hookleftarrow}^{1/2}$ terms never evaluate to Wrong. ◀

Equivalence and soundness imply that well-typed $\lambda^{1/2}_{\hookleftarrow}$ terms exhibit all the desired memory properties in the stack-based semantics. It is thus safe to "just not pop the stack":

▶ **Corollary 7** (Separation of environment and stack). *Evaluating well-typed* $\lambda^{1/2}_{\hookleftarrow}$ *terms never leaks stack references: If* $\mathsf{eval}^q_s\ k\ \mathsf{H}\ (\mathsf{S}, \Phi)\ t\ = (r,\ \mathsf{S}')$ *for well-formed* H *and* (S, Φ)*, then (1) all* H' *encountered during evaluation contain no stack pointers, and (2) if* $r = \mathsf{Done}\ (\mathsf{Val}\ v)$*, then all stack pointers in* v *are valid in* S'*.*

Hence, 2nd-class bindings can be safely implemented using a deferred stack discipline.

5 Implementation

We implement our system as an extension of Scala Native [50], a compiler backend for Scala that produces native code using LLVM [34]. Memory is managed at runtime by a non-copying variant of the Immix garbage collector [14]. We also implement our system in the MiniScala compiler used for teaching compiler classes at the authors' institution.

5.1 Scala Native

Scala Native compiles to LLVM which implements a fixed set of calling conventions and prohibits stack manipulation. Instead of allocating the `@stack` values on the system stack we rely on a *shadow stack*.

Shadow stack. Using a shadow stack simplifies the implementation, and allows us to implement the allocation scheme in any language and runtime. In addition, it optimizes memory usage of functions returning stack-qualified values, since only the returned value's storage is retained, while all other temporary stack values are reclaimed as usual.

The code generation is type directed but fairly simple: on entering functions that return a heap-qualified value we, instantiate a new stack frame in the shadow stack by first saving the previous top of the stack and then marking the new one. Shadow-stack allocations store the value in the current top of the stack, updating the stack pointer. When returning a heap-qualified value, the top of the shadow stack is reset to the top when entering the function. Since only frames for functions that return heap-qualified values are instantiated, calls to functions that return a stack-qualified value will update the stack pointer. In other words, our allocation strategy is similar to inlining regarding memory effects. Shadow-stack operations are inserted in a source Scala program as a source-to-source transformation. We insert calls for marking the top of the stack and resetting it.

Type system. We implement our type system as a compiler plugin for Scala's type system. The `@stack` qualifier is defined through the more general `@mode` annotation as shown below:

```
class mode[T] extends TypeConstraint
type stack = mode[Any];  type heap = mode[Nothing]
```

When type checking `T @mode[Q1] <: U @mode[Q2]` the type checker checks `T <: U` and `Q1 <: Q2` similarly to what is shown in the typing rules (Figure 3). Further implementation aspects (e.g., function signatures with qualifiers) closely follow those in [45]. The `@mode` annotation can be used to implement forms of storage-mode polymorphism (cf. Section 6.2).

5.2 MiniScala

MiniScala is a language and compiler which is used for teaching the compiler classes at the authors' institution. It differs from Scala Native in directly generating x86-64 assembly instead of LLVM, and in an overall *greatly* reduced feature set suitable for education, with a much simpler implementation of the type system and other components.

Native call stack. Following standard x86-64 conventions, the caller maintains a frame pointer (FP) pointing to the start of its own stack frame. The stack pointer (SP) points to the end of the stack, marking the point where fresh allocations can occur. Local variables are adressed FP-relative, i.e., as offsets of the frame pointer. Using this setup, a caller can treat a callee stack frame remaining after a call like any other piece of stack-allocated data, and continue allocating at the current stack pointer. Popping the caller stack frame will reset both FP and SP, and therefore reclaim all embedded callee stack frames, too.

Type system. In contrast to Scala Native, which can infer types based on bidirectional constraint propagation and resolution [44], MiniScala relies exclusively on a straightforward bidirectional typing algorithm without constraint generation. The simplicity of the algorithm means that generally more type annotations are required in user code than in full Scala.

Storage-mode polymorphism. MiniScala uses a simple erasure implementation of generics, and supports storage-mode polymorphism using allocator functions, but does not implement specific support to manage stack growth in generic code paths. Runtime dispatch on storage-mode witnesses could be added with reasonable implementation effort (cf. Section 6.2).

6 Discussion and Extensions

6.1 Tail calls

The stack-based semantics (Figure 5) does not model tail calls for simplicity and uniformity, i.e., function calls always create a fresh stack frame on top of the current one (EAppH and EAppS). However, it is possible to propagate tail contexts together with the qualifier q. Tail contexts calling a closure with a 1st-class argument can reuse the stack from its creation time, which is accessible from the captured stack pointer. Thus, we retain constant-space tail calls. This is safe since (1) all captured values at the closure's creation time are present, and (2) 1st-class parameters do not capture stack bindings. Tail calls with 2nd-class arguments cannot be optimized this way, as the argument may have been allocated anywhere on the stack, including in the current frame.

6.2 Storage-Mode Polymorphism

Basic subtyping. The subtyping relationship `@heap <: @stack` already provides *some* degree of polymorphism, allowing us to call functions accepting `@stack` parameters with `@heap` arguments. This encourages annotating non-escaping arguments as `@stack` since callers can pass both `@heap`- or `@stack`-qualified values. But naive subtype polymorphism has drawbacks: If we declare a function to return `@stack`, then (1) this does not guarantee that the return value is actually allocated on the stack, and (2) function returns cannot pop the stack anymore. It is thus desirable to introduce a proper notion of storage-mode polymorphism.

Parametric polymorphism. Having additional degrees of polymorphism is useful for two reasons: (1) dealing with higher-order functions, and (2) parameterizing where specific values are allocated. For example, we would like to generalize vector addition (Figure 1d) so that its result is allocated on the stack or on the heap. For case (1), we can use $F_{<:}$-style parametric polymorphism for storage modes, which is readily available in Scala's type system. We rephrase `@heap` and `@stack` as `@mode[Q]`, where `Q` ∈ `{Heap, Stack}`. For example, consider the higher-order function `logged`:

```
def logged[Q](f: (Int => Int) @mode[Q]): (Int => Int) @mode[Q] =
  { x => val res = f(x); println(res); res }
```

```
// Storage-mode polymorphic vector addition (type signature):
def add[Q](v1: Vec[f32] @stack, v2: Vec[f32] @stack): Vec[f32] @mode[Q] =
  { val vout = new Vec[f32](v1.length); for (i <- v1.length) { vout(i) = v1(i) + v2(i) }; vout }
```

```
def addH(v1: Vec[f32] @stack,
         v2: Vec[f32] @stack) =
{
  markStack()
  val vout = allocH(v1.length)
  ...
  resetStack()
  vout // : Vec[f32] @heap
}
```

(a) Monomorphization (heap)

```
def addS(v1: Vec[f32] @stack,
         v2: Vec[f32] @stack) =
{
  // do not mark stack
  val vout = allocS(v1.length)
  ...
  // do not reset stack
  vout // : Vec[f32] @stack
}
```

(b) Monomorphization (stack)

```
def add[Q](v1: Vec[f32] @stack,
           v2: Vec[f32] @stack,
    implicit m: Storage[Q]) = {
  m.markPoly()
  val vout = m.alloc(v1.length)
  ...
  m.resetPoly()
  vout // : Vec[f32] @mode[Q]
}
```

(c) Dynamic dispatch (witness)

Figure 6 Achieving polymorphic allocation and stack growth behavior at runtime via (a,b) monomorphization, (c) dynamic dispatch with a storage-mode witness parameter. Exception return paths are elided in all three versions. Dynamic dispatch can be achieved using object or function indirection as shown, or, defunctionalized and inlined, as conditional dispatch on a single tag bit.

The type parameter `Q` abstracts over the storage mode in the function's signature, and we can refer to it using the `@mode[Q]` annotation. Reusing Scala's type language for storage-mode polymorphism permits expressing more complex relationships among qualifiers. In the example below, the subtype bound on `Q3` requires it to subsume both `Q1` and `Q2`:

```
def compose[Q1, Q2, Q3 >: Q1 with Q2](
  f1: (Int => Int) @mode[Q1], f2: (Int => Int) @mode[Q2]
): (Int => Int) @mode[Q3] = x => f2(f1(x))
```

It is also possible to avoid explicit qualifier constraints altogether and have a type-system extension that infers the constraints from the body and checks them at call sites.

Parametric polymorphism for 2nd-class values has been studied by Osvald et al. [45], and we can build on their $D_{<:}^{1/2}$-calculus, a variant of $D_{<:}$, which is a core calculus for a subset of Scala [3, 2, 48]. This system can encode $F_{<:}$ with 1st-class type values and path-dependent types. Their encoding carries over to our setting, resulting in an analogous $D_{<:}^{1/2 \hookleftarrow}$-calculus [61]. This system abstracts over types and qualifiers separately.

Stack growth in generic code paths. Some operational aspects of storage-mode polymorphism require careful consideration. For a polymorphic return storage `@mode[Q]`, what code should a compiler generate? Keep or pop the function's stack frame when it returns? Consider a call tree of storage-polymorphic functions. If we instantiate `Q = Stack` we would expect the stack to grow, but what about `Q = Heap`?

There are three options: (1) no popping at all, (2) popping when polymorphic code returns to monomorphic code, and (3) popping throughout. A standard erasure implementation forces option (1), since only an instantiation `Q = Stack` can be assumed. Option (2) requires "compensation code" at call sites of parametric functions, delegating some of the stack-popping machinery to the caller instead of the callee. Option (3) can be achieved through monomorphization (i.e., compile-time specialization), or by tracking a runtime witness for each storage-mode parameter, making the popping decision via runtime dispatch.

Monomorphization. This requires generating different versions of the parametric function for each distinct instantiation it is called with. Figures 6a, 6b show the specialized version of `add` for heap respectively stack storage mode. The code is ideal since there is no overhead and stack management code can be generated for option (3), at the cost of a blowup in code size.

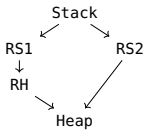
Parametric stack/heap allocation with runtime witness. To avoid the issues with monomorphization one can instead create runtime witnesses for storage modes and pass those as extra

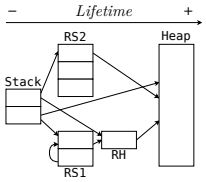
arguments. While avoiding code duplication, some dispatch overhead is induced at runtime. We can also abstract over allocation policies in a storage-polymorphic way. Figure 6c abstracts over different allocators and whether the result buffer is on `add`'s stack frame or on the heap. Passing an allocator works for explicit data structures (e.g., arrays), but closure allocation requires compiler support.

6.3 Levels Beyond Stack and Heap: Tracking Effect Capabilities

We inherit Osvald et al's. [45] ability to include arbitrary lattices between 1st class (bottom) and 2nd class (top) via subtyping and phantom types [35]. Operationally, this type hierarchy can be interpreted as encoding *lattices of stacks*, for example:

```
type Heap  = Nothing   // 1st class
type Stack = Any       // 2nd class
type RS1  <: Stack     // phantom types
type RS2  <: Stack
type RH   <: RS1
```





Closures at a given level ℓ close over values at the same or a descendant level, and popping the ℓ stack pops all its descendant stacks. It would be an interesting extension to map individual storage levels to separate physical runtime stacks. But even in the absence of such runtime support, there are interesting use cases of fine-grained lattices for structuring application code, including user-managed stack/heap region allocation policies.

Beyond memory management, our $\lambda_{\hookleftarrow}^{1/2}$-calculus supports programming with capabilities, and forms of lightweight effect polymorphism as proposed by [45], such as the familiar `withFile` example in Figure 7. Our system is strictly more expressive, as we support returning 2nd-class values. Lack of this ability impedes composability in practice, e.g., it is not possible to partially apply functions to capabilities. This issue inspired various solutions in related calculi [16, 43, 17, 15]. This paper shows that it can also be solved soundly and effectively with a small adjustment to the original system, namely allowing functions to return 2nd-class values. For example, the system now supports lazy collections in 2nd-class contexts:

```
withFile("foo.txt") { file =>
  val it = List(1,2,3,4).iterator.map(n => n + file.read().toInt) // iterator capturing file
  it.next() + it.next()  // ok (rejected in Osvald et al.'s system)
}
```

Finally, the example in Figure 7 also shows uses of nested scopes with *different* privilege levels, distinguishing read from write in the nested `parReduce` block, while maintaining the scoping guarantees of `withFile`. We have not seen evidence how other related works (e.g., capture types [15, 43] or reachability types [10]) can support this example with the same guarantees, so we pose it as a design challenge for works that seek to incorporate lightweight capability systems into realistic languages.

6.4 Stack References in Mutable Data Structures

One restriction we inherit from Osvald et al.'s system [45] is that 2nd-class values cannot be stored in mutable references/variables. Clearly, we do not want to store a stack reference in a heap-allocated mutable variable. But what about on-stack mutable variables? We have to be careful, since a reference could escape and be used after the containing stack frame is deallocated, e.g., if it was assigned to a variable further up the stack. However, assigning to on-stack variables is safe as long as the right-hand side was pushed on the stack before the variable. The key safety invariant is that stack references may only point upward, not downward. While a full solution for comparing lifetimes of variables seems possible using

```
// Capabilities: 1 <: R <: RW <: 2
type CanR  = CanRead  @mode[R]
type CanRW = CanWrite @mode[RW]

// File I/O:
trait File(val path: String) {
  def read(implicit c: CanR): Byte
  def write(implicit c: CanRW): Unit
}
```

```
def withFile[U](...)(f: (File @mode[R] => U) @stack): U
def parReduce[U](...)(f: ((U, U) => U) @mode[R]): U
withFile("foo.txt") { f =>
  parReduce(1 to 1000) { (a, b) =>
    f.read()  // ok:    'R' can capture 'R'
    f.write() // error: 'R' can't capture 'RW'!
  }
  f.write()   // ok:    '2' can capture 'RW'
}
```

Figure 7 Tracking effect capabilities (example from [45]). Files support `read` and `write` methods requiring implicit capabilities, arranged in a privilege lattice; `withFile` provides a non-escaping file to a block which has full privileges. The nested `parReduce` block has only reading privileges.

ownership [31, 59, 18, 41] or reachability types [10], this would add significant complexity. Still, partial solutions are possible using local reasoning on a best-effort basis, e.g., permitting assignments to variables within the same function.

Example: on-stack coroutines. Figure 8 shows a lowering transformation of a high-level coroutines API (adapted from [54]) to a stack-based implementation without heap allocations. To the left is the initial program in direct style, where a `decoder` continuously decompresses and feeds a character stream to a `parser` coroutine, encoded by delimited control operators `shift/reset` [21]. The `getChar` function for requesting the next character suspends the parser and transfers control back to the decoder, storing the parser's continuation in a shared mutable reference `emit`, which is invoked by the decoder once the next character is available. The shared state and the reified continuation should move from the heap onto the stack.

A key question is how to track the lifetimes of the continuations. We generally do not know the dynamic extent and hence would require storage on the heap. However, in this case we know the lifetime of the `emit` variable and may thus store the continuations on the stack.

The final concern is avoiding stack overflows from continued invocations of `emit`, and indeed it is possible to let this program run in *constant* stack space. This requires a few standard transformations (Figure 8, right), i.e., inlining `getchar`, compiling the nested loops (① and ②) into mutually tail-recursive functions, and a selective CPS conversion for `shift/reset` using Cong et al.'s [19] technique based on 2nd-class functions.

6.5 Use-Site Driven Inference of Storage Modes

Passing a function expression to a higher-order function that expects a (potentially) stack-allocated closure will cause the closure to be indeed allocated on the stack:

```
map(list, v => v + 10)                  // stack allocated
```

However, pulling out the function into a separate definition will cause it to be heap allocated:

```
val f = (v: Int) => v + 10              // heap allocated
map(list, f)
```

Unless the definition is marked explicitly for stack allocation:

```
@stack val f = (v: Int) => v + 10       // stack allocated
map(list, f)
```

It is easy to propagate storage-mode information from uses to definitions in a local scope, and automatically convert local declarations to stack allocation that would otherwise default to heap allocation, if they are never used in a truly 1st-class way. This analysis can be implemented in a single pass, without fixpoint computation, if the definitions are already well-typed under the more general mode.

```
// Produce a stream of characters:
def decoder() {
  while (true) {
    var c = getRawChar()
    if (c == EOF) break
    if (c == 0xFF) { // decompress
      var len = getRawChar()
      c = getRawChar()
      while (len > 0)
        emit(c) // transfer control to parser
      len -= 1
    } else
      emit(c)   // transfer control to parser
  }
  emit(EOF)     // transfer control to parser
}
// Consume characters:
def parser(): Unit = reset {
  var c: Char = '_'
  while (c != EOF) { ①
    c = getchar()
    if (c != EOF) {
      if (c.isLetter) {
        do { ②
          addToToken(c)
          // transfer control to decoder
          c = getChar()
        } while (c.isLetter)
        gotToken(WORD)
      }
      addToToken(c); gotToken(PUNCT)
    }
  }
}
parser(); decoder()
```

```
// Read a character from stdin:
def getRawChar() = StdIn.readChar()
// Store parser continuation:
@stack private var emit: Char => Unit = _
// Suspend and wait for decoder:
def getChar() = shift {
  (k: (Char => Unit) @stack) =>
    emit = k
}
// After inlining & selective CPS transformation:
def parser(): Unit @stack = {
  var c: Char = '_'
  var f1: (Char => Unit) @stack = null
  var f2: (Char => Unit) @stack = null
  def doWhileGetChar(c1: Char) = { ②
    c = c1; if (c.isLetter) dowhile()
  }
  def doWhile(): Unit =
    { addToToken(c); emit = f1 }
  def outWhileGetChar(c1: Char) = { ①
    c = c1
    if (c != EOF) {
      if (c.isLetter) {
        doWhile(); gotToken(WORD)
      }
      addToToken(c)
      gotToken(PUNCT)
    }
    outWhile()
  }
  def outWhile(): Unit = if (c != EOF) emit = f2
  f1 = x => doWhileGetChar(x)
  f2 = x => outWhileGetChar(x)
  outWhile()
}
```

Figure 8 On-stack coroutines example. Left: read, decompress, and parse data from a stream using coroutines `parser` and `decoder` (adapted from [54]). Right: transformation of `parser` into mutually recursive functions with stack-allocated closures. The red and blue parts belong to the transformed outer (①) and inner loop (②), respectively.

6.6 Function vs. Block Scope as Retention Boundary

For the most part of this paper, we have focused on functions as one particular control-flow construct, but it is also worth considering how stack growth and reclamation should behave in other block-scoped constructs, including loops.

Naively extending the lifetime of stack values to the function scope instead of the closest surrounding block scope would lead to exactly the same behavior as the `alloca` intrinsic in C. Notably, this would inhibit allocating stack values inside of a loop since it would quickly overflow the stack (left):

```
while (i < n) {
  @stack val tmp = new Vec[f32](10)
  use(tmp)
}
```

```
@inline def block(f: () => Unit) = f()
while (i < n) {
  block { ... }
}
```

Even without specific compiler support, it is possible to overcome this issue directly at the user level by using higher-order functions to denote block structure [32] (right). A block is simply a function that accepts a closure that is immediately applied.

Since the return type of `f` is `@heap` qualified, its stack-frame will be popped at the end of its body, making the program above run in constant stack space. Clearly, an equivalent transformation can be easily realized inside a compiler just as well.

```
// Example program:
fn f<F: FnOnce() -> i32>(g: F) {
  g();
}
let (t, mut s) = (1, 42);
f(|| { s += &t; s });

// Program after closure conversion:
fn f(g: Anon1) { g.call_once(()); }
let (t, mut s) = (1, 42);
f(Anon1 {s: &mut s, t: &t});
```

```
// Closure representation:
struct Anon1<'a>{
  s: &'a mut i32,
  t: &'a i32
}
impl<'a> FnOnce<()> for Anon1<'a> {
  type Output = i32; extern "rust-call"
  fn call_once(self, _unit: ()) -> i32 {
    *self.s += self.t;
    *self.s
} }
```

Figure 9 Closure conversion in Rust.

```
// OK: returned stack-allocated closure
fn captureByV<F: Fn(i32) -> i32>(f: F)
  -> impl Fn(i32) -> i32 {
    move |v| f(v) + 10
}
// Error: the returned closure is a union
fn cond(a: i32, b: i32) -> impl Fn(i32) -> i32 {
  if (b != 0) { |v| a * v + b }
  else { |v| a * v }
}
```

```
// Error: impl in nested position ①
fn spicy_curry() -> impl Fn(i32) ->
  (impl① Fn(i32) -> i32) {
    |a| move |b| a + b
}
// Error : recursive closure requires boxing
fn omega(i: i32) -> impl Fn(i32) -> i32 {
  let res = omega(i - 1);
  move |v| res(v) + 1
}
```

Figure 10 Valid and invalid returns of stack-allocated closures in Rust.

6.7 Stack Allocation for Closures and Other Anonymous Structures

A crucial application of storage mode qualifiers is in the context of closures and more generally in returning anonymous structures implementing a specific trait. In both cases, the absence of a concrete type at the call-site inhibits pre-allocating space on the caller's stack frame for the returned value. In this section, we investigate the challenges of compiling traits/interfaces for stack allocations in modern languages with stack environments like Rust.

Closure expansion in Rust. Figure 9 shows an example program in Rust, where a function `f` calls its closure parameter `g`, along with the program's closure conversion. The concrete argument for `f` is an anonymous closure that captures two stack-allocated variables by reference. First, note that `f` is monomorphized for the specific closure type which will be passed by value. Second, the closure desugars into the struct `Anon1`, where its fields represent the captured environment, and the method `call_once` represents the closure's body.

Limitations. To support stack allocation of closure objects (`structs`) the Rust compiler must be able to compute the size of the closure environment statically. In Figure 9, this is made possible by monomorphization, e.g., `Anon1` has the size of two `i32` references (the captured variables). This treatment of closures restricts their uses in return position. When generating code for a call to a function that returns a stack-allocated closure, the compiler has to reserve space for the returned closure in the caller's frame. However, it cannot know the size, because function types do not convey anything about the result's captured environment.

To mitigate this issue, Rust introduced abstract return types, which allow returning anonymous stack-allocated `structs` (e.g., `captureByV` in Figure 10). However, this feature comes with its own limitations — only a value of a single concrete type can be returned. The concrete return type is only superficially anonymous: the compiler tracks it to compute the required size for the stack allocation. This disallows certain programs (Figure 10), where the returned closure is data dependent (`cond`), or when a stack-allocated closure is returned from another closure (`spicy_curry`). The first problem can be addressed by pessimistically preallocating the size of the maximum closure that will be returned. The second one cannot be solved — the Rust compiler relies on the fact that it can always identify the concrete

underlying type for any `impl` type. Allowing it in nested positions breaks this property. Another problem is that captured environments can be recursive (`omega` in Figure 10) in which case Rust requires a boxed representation on the heap.

Storage-modes solution. All the examples in Figure 10 are supported by our system. For example, the `spicy_curry` definition can be implemented as

```
val curry: (i32 => (i32 => i32) @stack①) @stack = a => b => a + b
```

The `@stack` qualifier in the return type of a closure (①) renders it impossible to identify the size of the returned value at compile time, unless a whole-program analysis is performed. This is not at all required with storage modes, e.g., the following definition allows both separate compilation and partial applications yielding closures of unknown size:

```
// Module 1:
def f(g: (i32 => (i32 => i32)@stack)): i32 =
  // Closure of unknown size:
  val g' = g(10)
  g'(2)

// Module 2:
f(v1 => v2 => v1 + v2) // ok
```

```
// Module 3:
def incBy(v: i32): (i32 => i32) @stack =
  if (v == 0) { x => x }
  else { // recursive closure
    val rec = incBy(v - 1)
    { x => rec(x) + 1 }
  }
f(incBy(_)) // ok
```

Closures are not the only instance where storage modes are effective. Another one is in the context of lazy iterators, where a sequence of operations such as `fold ∘ map` would generate intermediate iterators which will eventually be consumed. The Rust implementation of such combinators relies on returning concrete monomorphized structures so that they can be stack allocated. Storage-mode qualifiers achieve this in a more straightforward manner:

```
def map[A, B](it: Iter[A] @stack, f: (A => B) @stack): Iter[B] @stack =
  new Iter[B] { def hasNext = it.hasNext;  def next() = f(it.next()) }
```

Finally, the stack semantics introduced by storage-mode qualifiers enables compilers to avoid treating `alloca` as an intrinsic and define it as a library function:

```
def alloca[T](n: Int): Array[T] @stack
```

7 Evaluation

In this section, we show how storage modes can improve memory management of larger programs. We first describe two case studies, differentiable programming and parser combinators, and then conclude with an in-depth evaluation of storage-mode-annotated programs.

7.1 Case Study: Differentiable Programming

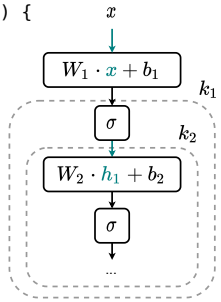
While storage modes are widely applicable for returning short-lived variable-size data, they are also useful for arena-style memory management. For example, computing gradients in differentiable programming requires keeping the intermediate tensors produced by operations in the forward pass of the dataflow graph until the operation node is traversed in the backward pass. The forward and backward pass are executed multiple times on different input tensors. Peak memory usage can be statically upper-bounded. However, the dataflow graph can be data dependent. Therefore, storage for intermediate tensors cannot be statically resolved. Instead, it has to be managed dynamically. Deep learning frameworks handle this by building a separate allocator with reference counting for managing the allocations in the graph.

Here we take a different approach, and implement a differentiable program using delimited continuations à la Wang et al. [58, 57]. Every primitive operation of the model (e.g., linear operators) takes a continuation denoting the rest of the forward and backward pass of the model. Figure 11 shows the definition of a linear layer that computes $l(x) = W_l \cdot x + b_l$.

```
class Linear(s: (Int, Int)) extends Layer {
  val W = Tensor.randn(s._1 :: s._2 :: Nil)                    // weights
  val b = Tensor.randn(s._1 :: Nil)                            // biases
  type TensorCPS = (Tensor @stack => Unit) @stack ③ => Unit
  def apply(x: Tensor @stack ①): TensorCPS @stack ② = { k =>
    val h = Tensor.matmuladd(W, x, b)                          // forward pass
    k(h)                                                       // ... continuation ...
    Tensor.∇matmuladd(h.∇, W, x, b)                            // backward pass
  }
}
```

Figure 11 Definition of a linear layer where h is allocated on the stack by `matmuladd` and is freed on returning along the backward pass.

```
class MLP(shapes: List[(Int, Int)]) {
  val (l1, l2, l3) = ...
  def apply(x: Tensor) =
    for {
      z1 <- l1(x)
      h1 <- σ(z1)
      z2 <- l2(h1)
      h2 <- σ(z2)
      o  <- l3(h2)
    }
    yield o
}
```




```
val mlp = MLP(...)
for (i <- 0 until epochs) {
  for ((x, y) <- data) {
    grad { x =>
      for {
        o <- mlp(x)
        ℓ <- loss(y, o)
      } yield ℓ
    }(x)
    pars.foreach(optimize(_, η))
  }
}
```

Figure 12 Fully differentiable program (multi-layer perceptron, MLP) with training loop. Intermediate values are stored on the stack and de-allocated eagerly during the backward pass. `MLP` is visualized in a dataflow graph.

Ignoring the `@stack` annotations, applying the linear layer first computes $W_l \cdot x + b_l$ through `matmuladd` producing the tensor h. It then calls the continuation which will eventually update the gradient field $h.\nabla$ of h and finally computes and accumulates the gradients for W and b.

Since the argument of function `apply` is annotated with `@stack` (①), its returned function of type `TensorCPS` must also be `@stack` (②). It cannot capture the argument otherwise. We also note that since the continuation `k` is used in a non-escaping fashion, it can be annotated as `@stack` (③). Finally, the tensor returned by `matmuladd` is also stack allocated. The full type of the returned function is `(((Tensor @stack => Unit)@stack)=> Unit)@stack`. Since its return type is not `@stack` annotated but its body calls a function that returns a `@stack`-annotated result, the compiler will insert marking and release instructions. This results in *every stack allocation happening in the execution of this function to be deallocated when returning.* This is safe because the continuation `k` uses the tensor `h` in a non-escaping fashion.

Figure 12 shows a fully differentiable program in our model. The `for` expressions are Scala's syntax for monadic comprehensions. Every temporary allocation performed by `l`i or σ is discarded after the backward operation of the respective layer as shown for the `Linear` layer before. The dataflow graph in the center depicts the `apply` function of the `MLP` class to the left. We represent continuations as dashed gray boxes. To the right is a program training an `MLP` instance. The program achieves arena-style memory management only through stack allocations and storage-mode qualifiers that ensure validity of references. The program is lightweight in annotations since those can be inferred given the definitions in Figure 11.

7.2 Case Study: Parser Combinators

The differentiable programming case study (Section 7.1) allocates most of the data on the stack (e.g., intermediate tensors). Yet, storage modes are also beneficial for interleaving stack

```
// Infix stack-function arrow:
type =@>[-A,+B] = (A => B) @stack

// On-stack parser type:
type SP[+U] = Parser[U] @stack

trait Result[+T] {
  def map[U](f: T =@> U): Result[U]
  def andThen[U](f: T =@> SP[U]): Result[U]
  def append[U >: T](alt: => Result[U]): Result[U]
}
```

```
@stack
class Parser[+T](val f: Input =@> Result[T]) {
  def flatMap[U](g: T =@> SP[U]): SP[U] =
    new Parser(f(_).andThen(g))
  def map[U](g: T =@> U): SP[U] =
    new Parser(f(_).map(g))
  def |[U >: T](p: => SP[U]): SP[U] =
    new Parser(in => f(in).append(p(in)))
  def ~[U](p: => SP[U]): SP[(T, U)] =
    this.flatMap(a => p.map(b => (a, b)))
}
```

Figure 13 On-stack parser combinators. Intermediate parser objects are short-lived. Making them stack-allocated reduces ephemeral heap allocations and fragmentation.

and heap allocation. An interesting example where this occurs is parser combinators.

Consider the definition of `Parser` in Figure 13. A parser produces a `Result[U]` which can be either `Failure` or `Success`. Parsers are built by combining other "base" parsers through combinators such as alternation `|` and sequencing `~`. For instance, the latter produces a parser that first applies a parser `f` on the input and then applies another parser `p` on the rest of the input producing two outputs. Note that the combinators' arguments are by-name which enables recursive parsers.

All `Parser` instantiations are allocated on the stack. This means that values closed over by parsers can also be marked as `@stack` (e.g., `g` in `flatMap`). To understand how the `@stack` annotations affect the computation, consider the following example that parses the strings matching the regular expression `(ab)*`: `def (ab)* = (lit("ab") ~ (ab)*) | lit("")`.

The `lit` combinator allocates a parser that recognizes the argument string. The recursive call is passed by name as argument to the `~` combinator, meaning that it will produce an allocation only when forced in the body of the newly built parser. Since parser functions have type `(Input => Result[T])`, the allocations of `Parser` happening on the stack will be reclaimed on return. During the evaluation of `(ab)*` on an input string, each recursive call will push a new parser object on the stack. Once the full input has been consumed, each stack-allocated parser object will be popped during the return path of the recursive calls.

Annotating `Parser` objects as `@stack` is beneficial in two ways: (1) it reduces the amount of ephemeral allocations on the heap, reducing overhead induced by collections, and (2) it reduces heap fragmentation. Running a parser results in alternating allocations of long-lived objects (the parsed results) and short-lived intermediate `Parser` objects. Many runtimes (e.g., Java and OCaml) handle this with generational garbage collection, allocating new objects in "young" regions and moving them to "old" regions once they stay reachable for long enough. In contrast, storage modes reduce heap fragmentation without relying on any assumption about the underlying heap memory management.

7.3 Performance Evaluation

We evaluate our storage-mode implementation in Scala Native (Section 5) on a range of benchmarks:

1. **filter**: This benchmark executes the code described in Section 2.2, which allocates the filtered list on the stack. The list size is 50 and 1/3 of the items are kept.
2. **flatten**: This benchmark flattens a binary tree of height 5 (32 nodes) by first building a tree of closures on the stack and then builds the list on the heap.
3. **curry**: A loop that curries a function.
4. **parser**: On-stack parser combinators, where parsed return values are heap allocated.

	AVG	AVG (`@stack`)	STD	STD (`@stack`)	AVG SPEEDUP
flatten (μs)	1.57	1.31	1.55	1.57	1.20x
mlp (ms)	859.27	702.12	2.99	4.58	1.22x
curry (μs)	22.14	20.24	6.82	4.94	1.09x
parser (μs)	707.25	563.68	59.94	79.95	1.25x
graph (ms)	32.66	28.14	3.39	2.10	1.16x
filter (μs)	86.43	77.78	16.48	14.29	1.11x

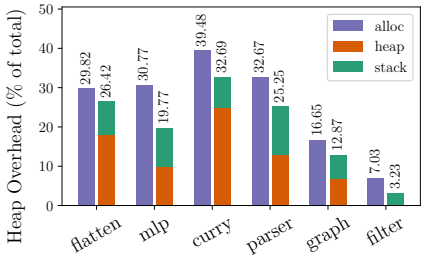


Figure 14 Run time comparison of annotated vs. unannotated programs. The chart shows the overhead of memory management as percentage of the total running time: "alloc" represents heap management in the unannotated version, and "heap + stack" represents heap and shadow stack management for the annotated version.

5. **graph**: Simulates insertions and deletions of edges and nodes on a graph. Insertions happen at a higher probability (55%) than deletions. At each insertion, the maximum distance between any 2 adjacent nodes is computed for the updated node. The graph is represented through adjacency lists on the heap. The maximum distance is computed using nested stack-allocated iterators.
6. **mlp** is the code shown in Section 7.1; most of the structures are stack allocated.

We compare each program annotated with `@stack` to its counterpart without `@stack` annotations. Everything else (compiler options, number of runs, etc.) remains unchanged. Benchmarks are compiled with Scala Native [50] generating LLVM byte code, which is then compiled through LLVM v10.0 with -O2. The benchmarks were run on a Intel Core i5-6300HQ CPU @ 2.30GHz on Ubuntu 20.04 LTS, Linux Kernel v5.11.0. All benchmarks stress the underlying memory management subsystem.

End-to-end runtime improvement. The table in Figure 14 shows averages and standard deviations of run times and the speedup for each benchmark. Each benchmark starts from a clean state and is run in a hot loop. The storage-modes version of each benchmark outperforms the unannotated version, and we observe average speedups of up to 25%.

Memory management overhead. The bar chart in Figure 14 shows the overhead percentage of memory management as computed through `perf v5.11.22` [28] with frequency `-F 30000`. Such a percentage is the number of times a memory management function activation record was alive during execution over the total samples of the stack. Bars on the left are percentages for the unannotated program, using the heap for most structures while bars on the right are the ones of the annotated program. Both percentages are scaled with respect to the running time of the unannotated program. The bars on the right of each benchmark are partitioned in the overhead for managing the heap (bottom) and the overhead for managing the stack (top). The size of these partitioned bars are also proxy metrics for the amount of allocations that happen on the stack for the annotated program. Overall, storage modes reduce the memory management overhead, since garbage collection is reduced.

Garbage collection behavior. Table 1 shows important statistics about the underlying memory management subsystem. First, we note that the number of times a collection is triggered in each benchmark is dramatically reduced. The median time for the marking and sweeping phases of each collection is shown in the table. The total time spent in marking and sweeping is more than halved, producing the performance improvements shown in Table 1.

An outlier is the increase in median time for marking in the graph benchmark. This is because a collection is triggered only after the heap is at maximum capacity. For the `@stack`

Table 1 Memory management evaluation. Columns show the number of GC executions, maximum shadow stack depth in bytes, time spent in the marking phase and sweeping phase.

	MODE	COLLECTIONS	STACK DEPTH (BYTES)	MARK MEDIAN (μs)	MARK TOTAL (ms)	SWEEP MEDIAN (μs)	SWEEP TOTAL (ms)
flatten	-	11126	0	16.79	200.82	15.33	177.20
	`@stack`	4950	2520	17.06	91.32	15.78	81.45
mlp	-	363	0	40.27	15.85	151.56	58.06
	`@stack`	116	18944	42.48	5.33	143.35	17.94
curry	-	1128	0	14.23	18.16	15.41	19.11
	`@stack`	342	40	14.76	5.87	15.04	5.62
graph	-	22	0	723.46	15.03	840.10	14.78
	`@stack`	14	416	1590.99	20.51	515.47	6.69
filter	-	967	0	16.65	18.16	16.29	17.01
	`@stack`	-	-	-	-	-	-
parser	-	51	0	21.36	1.15	15.66	0.88
	`@stack`	18	28448	21.58	0.49	16.40	0.33

version, this happens later in time, since ephemeral allocations due to iterators are stack allocated. At this point, the heap-allocated graph is larger and marking is more costly.

Currently, shadow-stack allocations are implemented as calls to an external function. It would be possible to further improve performance by optimization passes that exploit the non-escaping behavior imposed by the type system. Also, the marking phase of the GC still has to traverse the shadow stack (similarly to the call stack) to find heap roots. Hence, languages without GC can benefit even more from storage modes. For example, Swift uses reference counting, and making closure arguments non-escaping yields major performance improvements [53], which prompted the designers to make closure arguments non-escaping by default. Our storage modes provide strictly more benefits: not only can arguments be non-escaping, but so can be function return values, and they may be of variable size.

8 Related Work

Our work is inspired by Osvald et al. [45], who argued in favor of reintroducing 2nd-class values in modern programming languages. Instead of relying on escape analysis, they introduce $\lambda^{1/2}$, a language with qualified types where variables can be restricted to 2nd-class status, ensuring non-escaping behavior. The typing rules ensure that 2nd-class values follow a *strict* stack discipline and have been employed to model scoped capabilities [45, 46, 17, 43], in compiler intermediate representations for efficient compilation [19], and as compiler directives to improve stack allocation of closures [6]. We remove the limitation of their type system by enabling the return of 2nd-class values, which increases their composability, e.g., allowing currying over 2nd-class values. To achieve that, we propose a provably sound, relaxed stack semantics. Finally, instead of using the type system only for capability checking on the JVM version of Scala, we employ it for stack allocation on Scala Native [50] and provide an evaluation regarding the performance benefits of the system.

The problem of implementing programming language environments through a stack or a heap dates back to the Algol and LISP communities in the 70s. The LISP community named the problem as the "Funarg problem" [39] and the Algol community named it "Retention vs. deletion strategy" [12]. These works investigated the problem of escaping stack references in the presence of higher-order functions. Fischer [24] proved that the retention and deletion strategies were equally expressive by providing a transformation later recognized as one of the discoveries of CPS [47]. Our work combines these ideas, dealing with stack references through storage modes and returning variable-size data by implementing a delayed "deletion" strategy inspired by the corresponding selective CPS transformation [19].

The implementation of the shadow stack resembles Obstacks [26], which are "bump" memory allocators where allocation and deallocation is managed by pointer adjustments. The Obstacks API can be used as an architecture-independent lowering target to implement shadow stacks, but it does not come with static guarantees of its own.

Banerjee and Schmidt [9] designed a static criterion to approximate the runtime shape of the environment and determine whether a term could be evaluated using a single-stack environment. Their work extends the "simple expression" work of Georgeff [27]. Appel and Shao [5] argued in favor of fully allocating activation records on the heap.

Our approach is the first to enable stack allocation of variable-size data in direct style and can be retrofitted into language implementations relying on stack environments. The effectiveness of stack allocation has been investigated, among others, by Baker in the implementation of Scheme [8]. Efforts to add stack allocation are currently underway in many languages, including C♯ [36], Swift [53], and OCaml [42, 30].

Closely related to our work are region-based systems as proposed by Tofte and Talpin [55, 56]. While storage modes can be translated to a region calculus, our implementation relies on a simpler type system and does not require designing the language around region-based memory management. Compared to region-based memory management with explicit annotations, our solution is lightweight in terms of annotations. To ameliorate the annotation burden, region calculi are equipped with region inference. Storage modes are intentionally explicit to provide the programmer control over the stack, but also integrate well with type inference in languages that support it. Our work shows benefits in terms of performance, especially for compiled languages that automatically manage memory through garbage collection or reference counting. Nonetheless, storage modes are also compatible with type systems based on principles of ownership and borrowing, such as Rust's [31, 59], and can be beneficial to support variable-size data when compiling for embedded systems without support for dynamic heap allocations.

9 Conclusions

This paper addresses the problem of returning variable-size data in languages with stack environments. Our approach relies on storage modes, which are lightweight type annotations that guide stack allocations and de-allocations. The evaluation of storage modes as implemented in the Scala-Native compiler shows that our approach is beneficial for both reducing heap fragmentation, GC overhead and improving spatial locality. Storage modes can be implemented in high-level languages such as Swift and Scala to reduce heap pressure or in low-level languages such as Rust to promote uses of abstractions without paying the heap penalty.

Acknowledgements We thank the anonymous reviewers for their insightful comments. This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, and DOE award DE-SC0018050.

References

1 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, pages 8–19. ACM, 2003. doi:10.1145/888251.888254.

2 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World*, volume 9600

of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016. doi:10.1007/978-3-319-30936-1_14.

3 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *POPL*, pages 666–679. ACM, 2017. doi:10.1145/3009837.3009866.

4 Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, 1987. doi:10.1016/0020-0190(87)90175-X.

5 Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *J. Funct. Program.*, 6(1):47–74, 1996. doi:10.1017/S095679680000157X.

6 Apple Inc. Closures — The Swift Programming Language (Swift 5.4), May 2021. URL: https://web.archive.org/web/20220501162412/https://docs.swift.org/swift-book/LanguageGuide/Closures.html.

7 Kenichi Asai and Chihiro Uehara. Selective CPS transformation for shift and reset. In *PEPM*, pages 40–52. ACM, 2018. doi:10.1145/3162069.

8 Henry G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *ACM SIGPLAN Notices*, 30(9):17–20, September 1995. doi:10.1145/214448.214454.

9 Anindya Banerjee and David A. Schmidt. Stackability in the simply-typed call-by-value lambda calculus. *Sci. Comput. Program.*, 31(1):47–73, 1998. doi:10.1016/S0167-6423(96)00040-8.

10 Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021. doi:10.1145/3485516.

11 Friedrich L. Bauer. The cellar principle of state transition and storage allocation. *IEEE Ann. Hist. Comput.*, 12(1):41–49, 1990. doi:10.1109/MAHC.1990.10004.

12 Daniel M. Berry. Block structure: Retention or deletion? (extended abstract). In *STOC*, pages 86–100. ACM, 1971. doi:10.1145/800157.805041.

13 Malgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Log.*, 9(1):6, 2007. doi:10.1145/1297658.1297664.

14 Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, pages 22–32. ACM, 2008. doi:10.1145/1375581.1375586.

15 Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondrej Lhoták, and Martin Odersky. Tracking captured variables in types. *CoRR*, abs/2105.11896, 2021. URL: https://arxiv.org/abs/2105.11896, arXiv:2105.11896.

16 Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–30, 2022. doi:10.1145/3527320.

17 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA):126:1–126:30, 2020. doi:10.1145/3428194.

18 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013. doi:10.1007/978-3-642-36946-9_3.

19 Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.*, 3(ICFP):79:1–79:28, 2019. doi:10.1145/3341643.

20 Olivier Danvy. Defunctionalized interpreters for programming languages. In *ICFP*, pages 131–142. ACM, 2008. doi:10.1145/1411204.1411206.

21 Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160. ACM, 1990. doi:10.1145/91556.91622.

22 Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *PPDP*, pages 162–174. ACM, 2001. doi:10.1145/773184.773202.

23 Edsger W. Dijkstra. Recursive Programming. *Numerische Mathematik*, 2(1):312–318, December 1960. doi:10.1007/BF01386232.

24 Michael J. Fischer. Lambda calculus schemata. In *Proving Assertions About Programs*, pages 104–109. ACM, 1972. doi:10.1145/800235.807077.

25 Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203. ACM, 1999. doi:10.1145/301618.301665.

26 Free Software Foundation, Inc. The GNU C Library–Obstacks. URL: https://web.archive.org/web/20220509075117/https://www.gnu.org/software/libc/manual/html_node/Obstacks.html/.

27 Michael P. Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Trans. Program. Lang. Syst.*, 6(4):603–631, 1984. doi:10.1145/1780.1803.

28 Brendan Gregg. Linux Perf Examples, 2020. URL: https://web.archive.org/web/20220509003430/https://www.brendangregg.com/perf.html.

29 Sten Henriksson. A brief history of the stack. Technical report, 2011.

30 Jane Street. Memory management, 2022. URL: https://web.archive.org/web/20220401080322/https://signalsandthreads.com/memory-management/.

31 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018. doi:10.1145/3158154.

32 Peter J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 1965. doi:10.1145/363744.363749.

33 James R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, California Univ., Berkeley, Dept. of Electrical Engineering and Computer Sciences, May 1989.

34 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.

35 Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122. ACM, 1999.

36 Microsoft. C♯ language reference – stackalloc expression, 2022. URL: https://web.archive.org/web/20220405070936/https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/stackalloc.

37 James S. Miller and Guillermo Juan Rozas. Garbage collection is fast, but a stack is faster. Technical report, Massachusetts Institute of Technology, 1994.

38 James H. Morris. A bonus from van Wijngaarden's device. *Commun. ACM*, 15(8):773, August 1972. doi:10.1145/361532.361558.

39 Joel Moses. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *ACM SIGSAM Bulletin*, (15):13–27, July 1970. doi:10.1145/1093410.1093411.

40 Lasse R. Nielsen. A selective CPS transformation. In *MFPS*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 311–331. Elsevier, 2001. doi:10.1016/S1571-0661(04)80969-1.

41 James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998. doi:10.1007/BFb0054091.

42 OCaml Community. OCaml discourse: Add support for stack allocation, 2021. URL: https://web.archive.org/web/20210125181231/https://discuss.ocaml.org/t/add-support-for-stack-allocation/7039.

43 Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondrej Lhoták. Safer exceptions for scala. In *SCALA@SPLASH*, pages 1–11. ACM, 2021. doi:10.1145/3486610.3486893.

44 Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *POPL*, pages 41–53. ACM, 2001. doi:10.1145/360204.360207.

45 Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*, pages 234–251. ACM, 2016. doi:10.1145/2983990.2984009.

46 Leo Osvald and Tiark Rompf. Rust-like borrowing with 2nd-class values (short paper). In *SCALA@SPLASH*, pages 13–17. ACM, 2017. doi:10.1145/3136000.3136010.

47 John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3):233–247, November 1993. doi:10.1007/BF01019459.

48 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *OOPSLA*, pages 624–641. ACM, 2016. doi:10.1145/2983990.2984008.

49 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*, pages 317–328. ACM, 2009. doi:10.1145/1596550.1596596.

50 Denys Shabalin. Scala native, 2015. URL: https://web.archive.org/web/20220318165107/https://scala-native.readthedocs.io/en/latest/.

51 Amir Shaikhha, Andrew W. Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing style for efficient memory management. In *FHPC@ICFP*, pages 12–23. ACM, 2017. doi:10.1145/3122948.3122949.

52 Jeremy Siek. Type Safety in Three Easy Lemmas, 2013. URL: https://web.archive.org/web/20220308042857/https://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.

53 Swift Community. Use stack allocation for Swift closures #21933. URL: https://github.com/apple/swift/pull/21933/#issuecomment-454980737.

54 Simon Tatham. Coroutines in C, 2000. URL: https://web.archive.org/web/20220428071140/https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html.

55 Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *POPL*, pages 188–201. ACM Press, 1994. doi:10.1145/174675.177855.

56 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.

57 Fei Wang, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *NeurIPS*, pages 10201–10212, 2018.

58 Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, 2019. doi:10.1145/3341700.

59 Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The Essence of Rust. *arXiv:1903.00982 [cs]*, August 2020. arXiv:1903.00982.

60 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

61 Anxhelo Xhebraj, Oliver Bračevac, Guannan Wei, and Tiark Rompf. What if we don't pop the stack? The return of 2nd-class values (extended version). Technical report, Purdue University, 2022.