Parallelism in a Region Inference Context

MARTIN ELSMAN and TROELS HENRIKSEN, University of Copenhagen, Denmark

Region inference is a type-based program analysis that takes a non-annotated program as input and constructs a program that explicitly manages memory allocation and deallocation by dividing the heap into a stack of regions, each of which can grow and shrink independently from other regions, using constant-time operations.

Whereas region-based memory management has shown useful in the contexts of explicit region-based memory management, and in particular, in combination with parallel execution of code, combining region inference with techniques for higher-order parallel programming has not been investigated.

In this paper, we present an implementation of a fork-join parallel construct suitable for a compiler based on region inference. We present a minimal higher-order language incorporating the parallel construct, including typing rules and a dynamic semantics for the language, and demonstrate type soundness. We present a novel effect-based region-protection inference algorithm and discuss benefits and shortcomings of the approach. We also describe an efficient implementation embedded in the MLKit Standard ML compiler. Finally, we evaluate the approach and the implementation based on a number of parallel benchmarks, and thereby demonstrate that the technique effectively utilises multi-core architectures in a higher-order functional setting.

CCS Concepts: • **Software and its engineering → Garbage collection**; **Functional languages**; **Parallel programming languages**; *Runtime environments.*

Additional Key Words and Phrases: Region Inference, Parallelism, Memory Management

ACM Reference Format:
Martin Elsman and Troels Henriksen. 2023. Parallelism in a Region Inference Context. *Proc. ACM Program. Lang.* 7, PLDI, Article 142 (June 2023), 23 pages. https://doi.org/10.1145/3591256

1 INTRODUCTION

Region-based memory management allows programmers to associate objects with so-called regions, which may be explicitly allocated and deallocated by the programmer. Region-based memory management, as it is implemented for instance in Rust [Aldrich et al. 2002; Jung et al. 2017] and extensions to Cyclone [Gerakios et al. 2010; Grossman et al. 2002], can be a valuable tool for constructing critical systems, such as real-time embedded systems [Salagnac et al. 2006]. In contrast to explicit region-based memory management, *region inference* takes a non-annotated program as input and produces as output a region-annotated program, including directives for allocating and deallocating regions [Tofte et al. 2004]. The result is a programming paradigm where programmers can learn to write region-friendly code by following certain patterns [Tofte et al. 2022].

Whereas region inference has shown to be a viable strategy for sequential program execution [Birkedal et al. 1996; Elsman and Hallenberg 2021; Hallenberg et al. 2002], using region inference in a parallel shared-memory context has not received much attention, despite its intuitive promise in guiding a runtime system in which regions of memory are private to a particular task. A simple thread library could expose the following interface to a programmer:

Authors' address: Martin Elsman, mael@di.ku.dk; Troels Henriksen, athas@di.ku.dk, Department of Computer Science, University of Copenhagen, Universitetsparken 5, Copenhagen, Denmark, DK-2100.

2475-1421/2023/6-ART142
https://doi.org/10.1145/3591256

```
signature SIMPLE_THREAD = sig
  type α t
  val spawn : (unit → α) → α t
  val get   : α t → α
end
```

Using this interface, the programmer may fork new threads with the `spawn` function and join (i.e., synchronise) threads using the `get` function. Unfortunately, from a region-inference perspective, this interface is problematic because the `spawn` function says nothing about when the spawned function no longer needs access to external resources, such as values captured in the closure and stored in regions that are external to the function.

In general, the main obstacle in supporting parallelism in a high-level functional programming language is memory management. One possibility is for an implementation to make use of only a single heap, which easily leads to allocation races, which can be quite expensive for programs that allocate often. An alternative is for each thread to maintain its own heap and then use dynamic techniques to promote live objects to the parent's heap when a thread is joined. Different variations have been investigated and the techniques are highly influenced by the complexity of dealing with mutable objects and higher-order functions [Sivaramakrishnan et al. 2020; Westrick et al. 2019].

MPL [Westrick et al. 2019], which is an extension of MLton [Weeks 2006] with support for Fork-Join parallelism, exposes parallelism, essentially, through a function `par`:

```
val par : (unit→α) → (unit→β) → α * β
```

When applied to two thunks, `f` and `g`, the `par` function will spawn two threads for executing `f()` and `g()` in parallel and synchronize on the result.

Whereas the `par` function makes explicit when threads synchronise, in the context of region inference, the two functions may write into the same regions. We may choose a simple and efficient non-locking allocation scheme when the effect system tells us that races cannot occur and a more complex atomic allocation scheme when no such guarantee can be established. The non-locking allocation scheme will apply to allocations in thread-private regions but also to "destination-passing" cases where a region is allocated by one thread and allocated into only by one other thread.

The region scheme that we consider is based on the stack discipline. Whenever e is some expression, region inference may decide to replace e with the term **letregion** ρ **in** e' **end**, where e' is the result of transforming the expression e, which includes annotating allocating expressions with particular region variables (e.g., ρ) specifying the region each value should be stored in. The semantics of the **letregion** construct is first to allocate a region (initially an empty list of pages) on the region stack, bind the region to the region variable ρ, evaluate e', which may allocate values in the region (and perhaps in other regions), and, finally, deallocate the region bound to ρ (and its pages).[1] Regions may be passed to functions at run time (i.e., functions can be region polymorphic) and may be captured in closures. Soundness of region inference ensures that a region is not deallocated as long as a value within it is used by the remainder of the computation.

In this paper, we present an alternative `spawn` function that gives the programmer the possibility of specifying the lifetime of running threads. This aspect allows region inference to infer a useful conservative approximation to the lifetime of values required by a thread. The construct can be used to encode other useful parallel constructs, including MPL's `par` function. The technique does not rely on dynamic promotion strategies, even for mutable objects, but is instead based on the property that the lifetime of objects are determined statically by region inference.

[1] The implementation in the MLKit Standard ML compiler uses a region page size of 8KiB. Region pages are allocated from the underlying system in chunks and maintained in a shared free-list.

We demonstrate the usefulness of the `spawn` interface by using it to implement a number of parallel algorithms and by demonstrating that the algorithms perform and scale well in practice.

1.1 Contributions

The contributions of this work are the following:

(1) We present a fork-join parallel construct suitable for a compiler based on region inference and a minimal higher-order language incorporating the construct. We give typing rules and a dynamic semantics for the language and demonstrate type soundness both for the core language and for a region-explicit language, which serves as a target for region inference.
(2) We present an effect-based *protection inference* algorithm that separates regions into those allocated into at most by one thread and those that require atomic allocation because they are potentially allocated into by multiple threads.
(3) We discuss benefits and shortcomings of the approach and present an efficient implementation, which constitutes the first implementation of parallelism support in the MLKit, a compiler for the Standard ML programming language.
(4) We evaluate the approach and the implementation based on a number of parallel benchmarks, and thereby demonstrate that the approach leads to a viable technique for effectively utilising multi-core architectures in a higher-order functional setting.

The study is performed in the context of the MLKit [Tofte et al. 2022], which generates native x64 machine code for Linux and macOS [Elsman and Hallenberg 1995] and implements a series of techniques for refining the representations of regions [Birkedal et al. 1996; Tofte et al. 2004], including a technique for dividing regions into stack allocated (bounded) regions and regions that are unbounded and therefore heap allocated.

The parallel benchmarks are executed and compared with versions of the benchmarks compiled with MLton [Weeks 2006], MPL [Westrick et al. 2019], and with a sequential version of the MLKit.

1.2 Outline

The paper is organised as follows. In Section 2, we present a basic region-friendly interface for fork-join parallelism in ML, demonstrate its practical usefulness, and give motivations for its features, seen from a region perspective. In Section 3, we present typing rules and a dynamic semantics for an internal language featuring the parallel functionality. We also show how the library implementation of the parallel constructs are compiled into internal language constructs. In Section 4, we present region typing rules and a dynamic semantics for a region-explicit version of the internal language. The language serves as a target for region inference and we discuss how various language extensions, including region and effect polymorphism, has an influence on parallelism. We also present the effect-based protection inference algorithm that serves to distinguish between regions that are potentially allocated into by multiple threads and regions that are allocated into by at most one thread. In Section 5, we discuss the concrete implementation, which extends the MLKit Standard ML compiler. In Section 6, we present alternative approaches to avoid region allocation races. In Section 7, we present experimental results. In Section 8, we describe related work, and in Section 9, we conclude.

2 BASIC PARALLEL CONSTRUCTS

From a programmer's point of view, parallelism is exposed through the following library interface and reference implementation:

```
signature THREAD = sig
  type α t
  val spawn : (unit→α) → (α t→β) → β
  val get   : α t → α
end
```

```
structure Thread :> THREAD = struct
  type α t = α
  fun spawn f k = k(f())
  fun get x = x
end
```

The spawn function takes a function f to spawn and a scope function k. The spawn function spawns a new thread t and executes the application f() inside the thread. Parallelism is now achieved by allowing an implementation to proceed evaluating k t, the continuation applied to the thread t. The continuation may call the function get to wait for the result of evaluating the thread expression f(). Moreover, if get is not called during the evaluation of the continuation, the continuation will wait for the termination of the thread before the spawn construct reduces to the result of the continuation.

The Thread module allows us to implement other useful functions:

```
fun spawnarg f a k = spawn (fn() ⇒ f a) k
fun par f g = spawn f (fn a ⇒ spawn g (fn b ⇒ (get a, get b)))
```

We can also choose to implement the par function differently by having it spawn only one thread:

```
fun par f g = spawn g (fn b ⇒ (f(),get b))
```

As a more elaborate example, consider a parallel version of Mergesort on integer lists. The function makes use of a utility function split that takes a list and returns a pair of lists of roughly the same length. It also makes use of a utility function merge that takes as argument a pair of sorted lists and returns a sorted list containing the elements of the two argument lists. The Mergesort function pmsort that we shall define takes as argument a *par-number*, which indicates the parallel resources available to a thread. Based on the par-number, the function will diverge into a sequential Mergesort, when all available parallel resources have been used. Here is the definition of pmsort:[2]

```
fun pmsort p [] : int list = []
  | pmsort p [x] = [x]
  | pmsort p xs = let val q = p div 2
                      val (l,r) = split xs
                      val (ls,rs) = if p<=1 then (pmsort q l, pmsort q r)
                                    else par (fn () ⇒ pmsort q l,
                                              fn () ⇒ pmsort q r)
                  in merge(ls,rs)
                  end
```

Region inference results in the following region-annotated version of the function, which makes use of MLKit's region-polymorphic recursion to allow for the lists returned by the local calls to pmsort to be stored in local regions:

```
fun pmsort ρ p [] = [] at ρ
  | pmsort ρ p [x] = [x] at ρ
  | pmsort ρ p xs = let val q = p div 2
                        region ρ1 ρ2
                        val (l,r) = split ρ1 ρ2 xs
                        val (ls,rs) = if p<=1 then (pmsort ρ1 q l, pmsort ρ2 q r)
                                      else par (fn() ⇒ pmsort ρ1 q l,
                                                fn() ⇒ pmsort ρ2 q r)
                    in merge ρ (ls,lr)
                    end
```

[2]For reducing the number of allocations and the overall memory usage, a more efficient Mergesort uses vectors.

Notice how list construction is annotated with the region into which the list is allocated. Notice also that region-annotated programs are printed using region declarations (**region** ρ_1 ρ_2), which may appear inside **let**-bindings together with **val** and **fun** declarations.

In the case of parallel Mergesort, region inference has done a perfect job in that each region is allocated into by at most a single thread. Unfortunately, we cannot in general assume that regions are not allocated into by multiple threads. For this reason, we shall make a particular effort at allowing multiple threads to allocate into the same region by making region allocation atomic (using a partly lock-free implementation of allocation), unless we can infer that a region is allocated into by at most one thread. To illustrate the need for allocation atomicity, consider the following implementation of a parallel map construct, which, at first, may seem to work well:

```
fun pmap (f:α→β) (xs:α list) : β list =
  let fun g nil k = k()
        | g (x::xs) k = T.spawn (fn () ⇒ f x)
                                (fn t ⇒ g xs (fn () ⇒ T.get t :: k()))
  in  g xs (fn () ⇒ nil)
  end
```

The function spawns a new thread for each element in the argument list before any result is demanded using the `get` function. This function works well if the argument function allocates mostly in regions that are local to the function, for instance, if it makes allocations only in private regions and returns an integer. However, if, for example, the argument function allocates its result in a region, all invocations of the function will allocate into the same region, which may cause allocation congestion as all threads will compete for the region allocation pointer. Although the runtime system will ensure atomicity of allocations, a programmer may benefit, through faster execution, from arranging that multiple threads only rarely allocate into the same region.

3 INTERNAL LANGUAGE

The external library functions `spawn` and `get` that are present in the `Thread` structure are compiled into internal language constructs of the form **letspawn** `x` : τT = e **in** e' and `get` e.

The semantics of the construct **letspawn** `x` : τT = e **in** e' is to evaluate the expression e in the thread bound to the handle `x` (of type $\tau\,T$), which is in scope in the expression e'. The thread is forced to be joined after e' is evaluated to a value (meaning that the construct will wait until the thread terminates). Within e', the `get` construct can be used to force a join and extract the thread result before exiting the scope of the **letspawn** binding.

The grammar for a minimalistic internal language featuring thread support looks as follows:

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \tau \to \tau' & \text{– types} \\
 & \mid & \tau\,T & \text{– futures} \\
v & ::= & d \mid \lambda x:\tau.e & \text{– values} \\
 & \mid & \langle v \rangle & \text{– package}
\end{array}
\qquad
\begin{array}{rcll}
e & ::= & x \mid v \mid e_1\,e_2 & \text{– expressions} \\
 & \mid & \textbf{letspawn}\ x:\tau\,T = e\ \textbf{in}\ e' & \text{– thread creation} \\
 & \mid & \textbf{get}\ e & \text{– wait for result}
\end{array}
$$

Basic types (τ) include the type `int` of integers, function types ($\tau \to \tau'$), and the type of *thread futures* ($\tau\,T$). Values include integer values (d), function values ($\lambda x{:}\tau\,.\,e$), and *thread packages* ($\langle v \rangle$), which represent threads that have terminated with a value v. Besides taking the form of a **letspawn** construct or a `get` construct, an expression may take the form of a variable, a value, or an application.

The typing rules for the constructs of the internal language are given in Figure 1. The typing rules are straightforward. Notice that in the typing rule for the **letspawn** construct, no restrictions enforce thread futures of type $\tau\,T$ not to escape the scope of the construct. At runtime, however, a thread future can escape the scope of the binding **letspawn** construct only once the thread has

Expression and value typing $\boxed{\Gamma \vdash e : \tau}$

$$\Gamma, x : \tau \vdash x : \tau \qquad \Gamma \vdash d : \texttt{int} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau\, T \vdash e' : \tau'}{\Gamma \vdash \textbf{letspawn}\ x : \tau\, T = e\ \textbf{in}\ e' : \tau'} \qquad \frac{\Gamma \vdash e : \tau\, T}{\Gamma \vdash \textbf{get}\ e : \tau} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \langle v \rangle : \tau\, T}$$

Fig. 1. Typing rules for the internal language.

Small-step reduction rules $\boxed{e \hookrightarrow e'}$

$$\begin{array}{rcll} (\lambda x : \tau . e)\, v & \hookrightarrow & [v/x]e & \\ \textbf{get}\ \langle v \rangle & \hookrightarrow & v & \\ \textbf{letspawn}\ x : \tau\, T = v\ \textbf{in}\ e & \hookrightarrow & [\langle v \rangle / x]e & \\ E^\Gamma[e] & \hookrightarrow & E^\Gamma[e'] & \text{if } e \hookrightarrow e' \text{ and } E^\Gamma \neq [.] \end{array}$$

Fig. 2. Small-step reduction rules for the internal language.

been fully evaluated. Notice also that values are typed in non-empty type environments, which allows for values (e.g, functions) to refer to threads that are bound in thread contexts.

3.1 Dynamic Semantics

In order to present the dynamic semantics for the language, we first define the grammar for *redexes* (r) and *evaluation contexts* (E^Γ), which are given as follows:

$$\begin{array}{rcl} r & ::= & \textbf{letspawn}\ x : \tau\, T = v\ \textbf{in}\ e' \\ & | & \textbf{get}\ \langle v \rangle \\ & | & (\lambda x : \tau . e)\, v \end{array} \qquad \begin{array}{rcll} E^\Gamma & ::= & [.] & (\Gamma = \cdot) \\ & | & \textbf{letspawn}\ x : \tau\, T = e\ \textbf{in}\ E^{\Gamma'} & (\Gamma = \Gamma', x : \tau\, T) \\ & | & \textbf{letspawn}\ x : \tau\, T = E^\Gamma\ \textbf{in}\ e & \\ & | & E^\Gamma\ e \mid v\ E^\Gamma \mid \textbf{get}\ E^\Gamma & \end{array}$$

Evaluation contexts E^Γ make explicit, through Γ, the type of variables bound to threads in encapsulating **letspawn** constructs. When E^Γ is an evaluation context and e is an expression, we write $E^\Gamma[e]$ to denote the expression formed by filling the hole $[.]$ in the context E^Γ with the expression e. A *redex* (r) is an expression of a form that immediately matches the left-hand side of a reduction rule (to be defined shortly). Evaluation contexts are defined to pinpoint possible redexes inside threads spawned with the **letspawn** construct as well as the possible redexes in the host expression. In other words, for a given expression e, there may exist multiple pairs of evaluation contexts E^Γ and expressions e' such that $e = E^\Gamma[e']$. This feature allows for a nondeterministic order of evaluation with respect to the evaluation steps performed by each thread.

The small-step reduction rules in Figure 2 take the form $e \hookrightarrow e'$. The rule for function application is standard. In the rules for the **letspawn** construct, we see that when a thread has terminated with a value v, the value is packaged into a thread package $\langle v \rangle$ and substituted into the thread's scope in place of the variable that the thread is bound to. The rule for the `get` construct allows for opening a thread package. The context rule allows for evaluation within a context.

3.2 Type Safety for the Internal Language

The type safety property that we shall prove for the internal language is based on well-known techniques for proving type safety [Morrisett 1995; Wright and Felleisen 1994]. We highlight the main properties below, with details provided in Appendix A.1 (auxiliary material).

The following property states that a well-typed expression is either a value or can be separated into an evaluation context and a redex, which is demonstrated by induction over the structure of e:

Proposition 3.1 (Decomposition). *If $\vdash e : \tau$ then either e is a value or there exist a redex e', a type τ', and a context E^Γ such that $\Gamma \vdash e' : \tau'$ and $e = E^\Gamma[e']$.*

Notice that Decomposition does not suggest that when e is not a value then it is constructed based on a unique redex and a unique evaluation context. Instead, Decomposition just suggests that such a pair of a redex and an evaluation context exists.

Natural properties about contextual typing and value substitution hold, which are demonstrated by induction over the structure of E^Γ and e, respectively:

Proposition 3.2 (Context). *If $\Gamma_0 \vdash E^\Gamma[e'] : \tau$ then $\Gamma_0, \Gamma \vdash e' : \tau'$ for some τ'. Further, if $\Gamma_0 \vdash E^\Gamma[e] : \tau$ and $\Gamma_0, \Gamma \vdash e : \tau'$ and $\Gamma_0, \Gamma \vdash e' : \tau'$ then $\Gamma_0 \vdash E^\Gamma[e'] : \tau$.*

Proposition 3.3 (Value Substitution). *If $\Gamma, x : \tau \vdash e : \tau'$ and $\Gamma \vdash v : \tau$ then $\Gamma \vdash [v/x]e : \tau'$.*

Type preservation (i.e., subject reduction) for the language, as well as progress, can be stated as follows and shown by induction over the structure of e and by use of Proposition 3.1:

Proposition 3.4 (Type Preservation). *If $\Gamma \vdash e : \tau$ and $e \hookrightarrow e'$ then $\Gamma \vdash e' : \tau$.*

Proposition 3.5 (Progress). *If $\vdash e : \tau$ then either e is a value or $e \hookrightarrow e'$ for some e'.*

No matter in which order redexes are evaluated, evaluation leads to the same result.

Proposition 3.6 (Confluence). *If $\Gamma \vdash e : \tau$ and $e \hookrightarrow^* e_1$ and $e \hookrightarrow^* e_2$ then there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow^* e'$.*

Proof. Based on Newmann's Lemma [Newman 1942], it suffices to show that if $\Gamma \vdash e : \tau$ and $e \hookrightarrow e_1$ and $e \hookrightarrow e_2$ then there exists e' such that $e_1 \hookrightarrow^* e'$ and $e_2 \hookrightarrow^* e'$. The proof proceeds by induction over the structure of e with the interesting case being the `letspawn` construct. □

3.3 Compilation

The library functions `spawn` and `get` that are present in the `Thread` structure are compiled into `letspawn` and `get` constructs in the internal language. The compilation is a straightforward syntactic translation of the program. We only present a few of the translation rules:

Compilation $\boxed{[\![e]\!] = e'}$

$$
\begin{array}{rcl}
[\![\texttt{spawn}\ f\ e]\!] & = & \texttt{letspawn}\ x : \tau\ T = [\![f()]\!]\ \texttt{in}\ [\![e\ x]\!] \\
 & & \quad \text{where } f : \texttt{unit} \to \tau \\
[\![\texttt{get}\ e]\!] & = & \texttt{get}\ [\![e]\!]
\end{array}
\qquad
\begin{array}{rcl}
[\![e\ e']\!] & = & [\![e]\!]\ [\![e']\!] \\
[\![x]\!] & = & x \\
 & \cdots &
\end{array}
$$

4 REGION-ANNOTATED PROGRAMS

In this section, we present the notions of region-annotated types and region-annotated expressions along with typing rules and a dynamic semantic. Region inference is the process of transforming a non region-annotated (internal language) expression into a region-annotated (internal language) expression. We shall not here describe the process of region inference in detail. However, we shall discuss the typing rules in details and demonstrate how the typing rules are proven sound with respect to a small-step dynamic semantics for the language. Moreover, by defining a straightforward notion of region erasure on terms, written $er(\cdot)$, it will also be straightforward to demonstrate an evaluation erasure property stating that if a region-annotated expression e evaluates to some (region-annotated) value v then the erasure of e evaluates to the erasure of v.

Region-annotated values $\boxed{\Gamma \vdash v : \tau}$

$$\Gamma \vdash d : \texttt{int} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \langle v \rangle^{\rho} : ((\tau, \varphi)T, \rho)} \qquad \frac{\Gamma, x : \tau' \vdash e : \tau, \varphi}{\Gamma \vdash \lambda^{\rho} x : \tau'.e : (\tau' \xrightarrow{\varphi} \tau, \rho)}$$

Region-annotated expressions $\boxed{\Gamma \vdash e : \tau, \varphi}$

$$\frac{\Gamma \vdash e : \tau, \varphi \quad \varphi' \supset \varphi}{\Gamma \vdash e : \tau, \varphi'} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau, \emptyset} \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \tau, \emptyset}$$

$$\frac{\Gamma \vdash e : \tau, \varphi \qquad \varphi'' = \varphi \cup \varphi' \cup \{\rho\} \quad \Gamma, x : ((\tau, \varphi)T, \rho) \vdash e' : \tau', \varphi'}{\Gamma \vdash \texttt{letspawn}\ x : ((\tau, \varphi)T, \rho) = e\ \texttt{at}\ \rho\ \texttt{in}\ e' : \tau', \varphi''} \ (1) \qquad \frac{\Gamma \vdash e : ((\tau, \varphi_0)T, \rho), \varphi}{\Gamma \vdash \texttt{get}\ e : \tau, \varphi \cup \{\rho\}} \ (2)$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau, \varphi}{\Gamma \vdash \lambda x : \tau'.e\ \texttt{at}\ \rho : (\tau' \xrightarrow{\varphi} \tau, \rho), \{\rho\}} \qquad \frac{\Gamma \vdash e : (\tau' \xrightarrow{\varphi_0} \tau, \rho), \varphi \qquad \Gamma \vdash e' : \tau', \varphi'}{\Gamma \vdash e\ e' : \tau, \varphi \cup \varphi' \cup \varphi_0 \cup \{\rho\}}$$

$$\frac{\Gamma \vdash e : \tau, \varphi \qquad \rho \notin \mathrm{frv}(\Gamma, \tau)}{\Gamma \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e : \tau, \varphi \setminus \{\rho\}} \qquad \frac{\Gamma \vdash e : \tau, \varphi \qquad \Gamma, x : ((\tau, \varphi)T, \rho) \vdash e' : \tau', \varphi'}{\Gamma \vdash \texttt{letspawn}\ x : ((\tau, \varphi)T, \rho) = e\ \texttt{in}\ e' : \tau', \varphi \cup \varphi'} \ (3)$$

Fig. 3. Typing rules for region-annotated values and region-annotated expressions.

We first define the notions of *effects* (φ), and *(region annotated) types* (τ), as follows:

$$\begin{array}{lcll} \varphi & ::= & \{\rho_1, \cdots, \rho_n\} & \text{– effect} \\ \tau & ::= & \texttt{int} & \text{– unboxed integer} \\ & | & (\tau_1 \xrightarrow{\varphi} \tau_2, \rho) & \text{– boxed function} \\ & | & ((\tau, \varphi)T, \rho) & \text{– boxed future} \end{array}$$

Effects are simply sets of region variables. We shall not here in this minimalistic treatment distinguish between the effect of allocating in a region (a so-called put-effect) or reading a value in a region (a so-called get-effect). Boxed types (both function types and futures) are associated with a region specifying where the values of the particular type are stored. Also, both function types and the type of futures are annotated with latent effects, which specify, respectively, the effect of evaluating the function and the effect of evaluating the thread future.

Region annotated expressions and region annotated values are defined as follows:

$$\begin{array}{lcll} e & ::= & x \mid v \mid \lambda x : \tau.e\ \texttt{at}\ \rho \mid e_1\ e_2 & \text{– expressions} \\ & | & \texttt{letregion}\ \rho\ \texttt{in}\ e & \text{– region creation} \\ & | & \texttt{letspawn}\ x : \tau = e\ \texttt{at}\ \rho\ \texttt{in}\ e' & \text{– thread creation} \\ & | & \texttt{letspawn}\ x : \tau = e\ \texttt{in}\ e' & \text{– thread context} \\ & | & \texttt{get}\ e & \text{– wait for result} \end{array} \qquad \begin{array}{lcll} v & ::= & d \mid \lambda^{\rho} x : \tau.e & \text{– value} \\ & | & \langle v \rangle^{\rho} & \text{– package} \end{array}$$

Values include unboxed integers (d), closures ($\lambda^{\rho} x : \tau.e$), and thread packages ($\langle v \rangle^{\rho}$). Closures and thread packages are boxed and associated with distinguished regions. An expression can be a variable (x), a value (v), a lambda expression, a function application ($e_1\ e_2$), a **letregion** construct, a **letspawn** allocation-construct, a **letspawn** context, or a `get` construct. Notice that allocating expressions (i.e., lambda expressions and **letspawn** allocation-constructs) are annotated with an **at** specifier, which specifies in which region the value (or thread object) should be allocated. The **letspawn** thread-context is used to capture intermediate thread states (e and e' may evaluate in parallel). The *free (program) variables* of some expression (or value) e is written $\mathrm{fpv}(e)$. We use Γ to range over *type environments*, which are finite maps from program variables to region types.

Allocation and Deallocation $\boxed{e \xhookrightarrow{\varphi} v}$

$$\lambda x : \tau.e \;\texttt{at}\; \rho \xhookrightarrow{\{\rho\}\cup\varphi} \lambda^{\rho} x : \tau.e$$
$$\texttt{letspawn}\; x : \tau = e \;\texttt{at}\; \rho \;\texttt{in}\; e \xhookrightarrow{\{\rho\}\cup\varphi} \texttt{letspawn}\; x : \tau = e \;\texttt{in}\; e$$
$$\texttt{letregion}\; \rho \;\texttt{in}\; v \xhookrightarrow{\varphi} v$$

Reduction and Context $\boxed{e \xhookrightarrow{\varphi} e'}$

$$(\lambda^{\rho} x : \tau.e)\, v \xhookrightarrow{\{\rho\}\cup\varphi} [v/x]e$$
$$\texttt{get}\; \langle v \rangle^{\rho} \xhookrightarrow{\{\rho\}\cup\varphi} v$$
$$\texttt{letspawn}\; x : ((\tau, \varphi')T, \rho) = v \;\texttt{in}\; e \xhookrightarrow{\varphi} [\langle v \rangle^{\rho}/x]e$$
$$E^{\Gamma}_{\varphi}[e] \xhookrightarrow{\varphi'} E^{\Gamma}_{\varphi}[e'] \quad \text{if } e \xhookrightarrow{\varphi\cup\varphi'} e' \text{ and } E^{\Gamma}_{\varphi} \neq [.] \text{ and } \varphi \cap \varphi' = \emptyset$$

Fig. 4. Dynamic semantics for region-annotated programs.

Typing rules for region-annotated values and region-annotated expressions are presented in Figure 3. Regarding rule (1), notice that the latent effect of the thread function is included in the type of the thread, which has the consequence that the regions in the latent effect are kept alive during the scope of the `letspawn` construct. Regarding rule (2), notice that waiting for a thread is not in itself associated with an effect besides reading from the boxed future-value. When dealing with the full language, which supports exceptions, calling `get`, however, may have the effect of raising an exception X, if the exception X was raised by the queried thread. Notice also that a programmer may return (or store somewhere, using mutable data structures) a closure that calls `get` and that the effect of such a call is only to access the value in the boxed future; if control has left the scope of the `letspawn` construct, the thread has terminated and the result is available and kept alive as long as the regions in which the result is present are alive.

4.1 Dynamic Semantics

In order to present the dynamic semantics for the region-annotated language, we first define the grammar for *redexes* (r) and *evaluation contexts* (E^{Γ}_{φ}), which are given as follows:

$$\begin{array}{lcll}
r & ::= & \texttt{letspawn}\; x : \tau = e \;\texttt{at}\; \rho \;\texttt{in}\; e \mid \texttt{letspawn}\; x : \tau = v \;\texttt{in}\; e & \\
 & \mid & \texttt{letregion}\; \rho \;\texttt{in}\; v \mid \texttt{get}\; \langle v \rangle^{\rho} \mid (\lambda^{\rho} x : \tau.e)\, v & \\
E^{\Gamma}_{\varphi} & ::= & [.] & (\Gamma = \cdot, \delta = \cdot) \\
 & \mid & \texttt{letspawn}\; x : \tau = e \;\texttt{at}\; \rho \;\texttt{in}\; E^{\Gamma'}_{\varphi} & (\Gamma = x : \tau, \Gamma') \\
 & \mid & \texttt{letspawn}\; x : \tau = e \;\texttt{in}\; E^{\Gamma'}_{\varphi} & (\Gamma = x : \tau, \Gamma') \\
 & \mid & \texttt{letregion}\; \rho \;\texttt{in}\; E^{\Gamma}_{\varphi'} & (\varphi = \{\rho\} \cup \varphi') \\
 & \mid & \texttt{letspawn}\; x : \tau = E^{\Gamma}_{\varphi} \;\texttt{in}\; e \mid E^{\Gamma}_{\varphi}\; e \mid v\; E^{\Gamma}_{\varphi} \mid \texttt{get}\; E^{\Gamma}_{\varphi} &
\end{array}$$

Evaluation contexts E^{Γ}_{φ} make explicit, through Γ, the type of variables bound to threads in encapsulating `letspawn` constructs, and, through φ, the region variables bound to regions in encapsulating `letregion` constructs. When E^{Γ}_{φ} is an evaluation context and e is an expression, we write $E^{\Gamma}_{\varphi}[e]$ to denote the expression formed by filling the hole $[.]$ in the context E^{Γ}_{φ} with the expression e.

The evaluation rules are given in Figure 4 and consist of *allocation and deallocation rules*, *reduction rules*, and a *context rule*. The rules are of the form $e \xhookrightarrow{\varphi} e'$, which says that, given a set of allocated regions φ, the expression e reduces (in one step) to the expression e'. Notice in particular how the

letspawn thread-creation construct reduces in one step to a **letspawn** thread-context with an effect that includes the region variable associated with the thread future.

We can now define an *evaluation* relation $\hookrightarrow^{\varphi*}$ as the least relation formed by the reflexive transitive closure of the relation $\hookrightarrow^{\varphi}$.

4.2 Type Safety for the Region-Annotated Internal Language

The proof of type safety is based on an earlier proof of type safety for a similar language, but a language that does not feature parallel execution of threads [Elsman 2003]. We shall therefore not present the complete proofs here, but instead report on the involved propositions and highlights of the proofs. The structure also closely resembles the structure of the type-safety proof for the non-region-annotated version of the internal language, as covered in Section 3.2. More details are available in Appendix A.2 (auxiliary material).

A well-typed expression is either a value or can be separated into an evaluation context and a redex, which is demonstrated by induction over the structure of e:

Proposition 4.1 (Decomposition). *If $\vdash e : \tau, \varphi'$ then either (1) e is a value or (2) there exists a redex e', a type τ', and a context E^{Γ}_{φ} such that $e = E^{\Gamma}_{\varphi}[e']$ and $\Gamma \vdash e' : \tau', \varphi \cup \varphi'$.*

Notice how the parameterisation of contexts allows us to establish a proper type environment for an inner expression. Following the development for the non-region-annotated internal language, natural properties about contextual typing and value substitution hold, and are demonstrated by induction over the structure of E^{Γ}_{φ} and e, respectively:

Proposition 4.2 (Context). *If $\Gamma_0 \vdash E^{\Gamma}_{\varphi}[e'] : \tau, \varphi'$ then $\Gamma_0, \Gamma \vdash e' : \tau', \varphi \cup \varphi'$ for some τ'. Further, if $\Gamma_0 \vdash E^{\Gamma}_{\varphi}[e] : \tau, \varphi'$ and $\Gamma_0, \Gamma \vdash e : \tau', \varphi \cup \varphi'$ and $\Gamma_0, \Gamma \vdash e' : \tau', \varphi \cup \varphi'$ then $\Gamma_0 \vdash E^{\Gamma}_{\varphi}[e'] : \tau, \varphi'$.*

Proposition 4.3 (Value Substitution). *If $\Gamma, x : \tau \vdash e : \tau', \varphi$ and $\Gamma \vdash v : \tau$ then $\Gamma \vdash [v/x]e : \tau', \varphi$.*

Type preservation and progress, can be stated as follows and shown by induction over the structure of e and by use of Proposition 4.1:

Proposition 4.4 (Type Preservation). *If $\Gamma \vdash e : \tau, \varphi$ and $e \hookrightarrow^{\varphi} e'$ then $\Gamma \vdash e' : \tau, \varphi$.*

Proposition 4.5 (Progress). *If $\vdash e : \tau, \varphi$ then either e is a value or $e \hookrightarrow^{\varphi} e'$ for some e'.*

4.3 Language Extensions

The region-annotated internal language that we have introduced in the previous sections demonstrate the overall soundness of the approach of using region-based memory management in a parallel context, but is overly minimalistic as a basis for an implementation of a real programming language, such as Standard ML. It is straightforward to add features such as tuples, records, conditionals, parameterised sum-types, exceptions, and even recursive functions to the language. It is also well-known how region inference can support Hindley-Milner style let-polymorphism, region- and effect-polymorphic functions, and even region-polymorphic recursion.

We now outline how the language is extended with region- and effect-polymorphic functions and polymorphic recursion, which allows us to discuss techniques for avoiding allocation races based on region polymorphism. We also discuss aspects of adding mutable data (i.e., references and arrays) to the language, as well as how region representation analyses [Birkedal et al. 1996], such as multiplicity analysis, are affected by the addition of parallelism.

Region- and Effect-Polymorphic Functions. We shall use ϵ to range over so-called *effect variables*. An effect is now a set of *basic effects* (η), each of which can be either an effect variable or a region

variable. Latent effects in the types of boxed functions and boxed futures, are now of the form $\epsilon.\varphi$. Such objects are called *arrow effects* and are central when it comes to defining the notion of unification, which is the foundation for the region inference algorithm that we build upon [Tofte and Birkedal 1998, 2000]. Here are the (refined) definitions of basic effects, effects, types, and type schemes, which are types parameterised over type variables, effect variables, and region variables:

$$
\begin{array}{lcll}
\tau & ::= & \alpha & \text{– type variable} \\
 & | & \texttt{int} & \text{– unboxed integer} \\
 & | & (\tau_1 \xrightarrow{\epsilon.\varphi} \tau_2, \rho) & \text{– boxed function} \\
 & | & ((\tau, \epsilon.\varphi)T, \rho) & \text{– boxed future}
\end{array}
\qquad
\begin{array}{lcll}
\eta & ::= & \epsilon \mid \rho & \text{– basic effect} \\
\varphi & ::= & \{\eta_1, \cdots, \eta_n\} & \text{– effect} \\
 & & & \\
\sigma & ::= & \tau \mid \forall \vec{\alpha}\vec{\epsilon}\vec{\rho}.\sigma & \text{– type scheme}
\end{array}
$$

A *substitution* (S) is a triple $(S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$, where S^{r} is a *region substitution*, mapping region variables to region variables, S^{t} is a *type substitution* mapping type variables to types, and S^{e} is an *effect substitution*, mapping effect variables to arrow effects. The effect of applying a substitution on an object is to perform the three substitutions simultaneously on the three kinds of variables in the object (by renaming of bound variables within the object to avoid capture). For effects and arrow effects, substitution is defined as follows [Tofte and Birkedal 2000], assuming $S = (S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$:

$$
\begin{aligned}
S(\varphi) &= \{S^{\mathrm{r}}(\rho) \mid \rho \in \varphi\} \cup \{\eta \mid \exists \epsilon, \epsilon', \varphi' \text{ s.t. } \epsilon \in \varphi \wedge S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi' \wedge \eta \in \{\epsilon'\} \cup \varphi'\} \\
S(\epsilon.\varphi) &= \epsilon'.(\varphi' \cup S(\varphi)), \text{ where } S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi'
\end{aligned}
$$

We can now appreciate why the types of boxed functions and boxed futures are annotated with arrow effects $\epsilon.\varphi$ and not with effects φ. With arrow effects, if a non-region-annotated type is given two distinct region annotations, then there exists a *unifier* (i.e., a substitution) that, when applied to the two types, makes the two resulting region-annotated types equal, which is essential for the unification-based region inference algorithm [Tofte and Birkedal 1998].

A type scheme $\sigma = \forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\tau'$ *generalises* a type τ *via* $\vec{\rho}\,'$, written $\sigma \geq \tau$ via $\vec{\rho}\,'$, if there exists a substitution $S = (\{\vec{\rho}\,'/\vec{\rho}\}, S^{\mathrm{t}}, S^{\mathrm{e}})$ such that $S(\tau') = \tau$, $\mathrm{dom}(S^{\mathrm{t}}) = \{\vec{\alpha}\}$, and $\mathrm{dom}(S^{\mathrm{e}}) = \{\vec{\epsilon}\}$. If $\sigma \geq \tau$ via $\vec{\rho}$, for some σ, τ, and $\vec{\rho}$, and S is a substitution, then $S(\sigma) \geq S(\tau)$ via $S(\vec{\rho})$.

Using the above definitions, typing rules for function application and for recursive functions supporting type polymorphism, effect polymorphism, and region polymorphism (and region-polymorphic recursion) can be given as follows, following [Elsman and Hallenberg 2021]:

$$
\frac{\Gamma \vdash e : \sigma, \varphi \qquad \Gamma \vdash e' : \tau_1, \varphi_1 \qquad \sigma \geq (\tau_1 \xrightarrow{\epsilon.\varphi_0} \tau_2, \rho) \text{ via } \vec{\rho}}{\Gamma \vdash e\ [\vec{\rho}]\ e' : \tau_2, \varphi \cup \varphi_1 \cup \varphi_0 \cup \{\rho\}}
\qquad
\frac{\sigma = \forall \vec{\rho}\vec{\epsilon}.(\tau_1 \xrightarrow{\epsilon.\varphi} \tau_2, \rho) \qquad \Gamma + \{f : \sigma, x : \tau_1\} \vdash e : \tau_2, \varphi \qquad \mathrm{fv}(\vec{\alpha}\vec{\epsilon}\vec{\rho}) \cap \mathrm{fv}(\Gamma, \varphi, \rho) = \emptyset}{\Gamma \vdash \textbf{fun}\ f\ [\vec{\rho}]\ x = e\ \textbf{at}\ \rho : \forall \vec{\alpha}.\sigma, \{\rho\}} \quad \boxed{\Gamma \vdash e : \sigma, \varphi} \tag{4}
$$

We shall not here present the typing rule for the associated value version of recursive function closures, which is necessary for proving soundness (along with appropriate substitution lemmas), but emphasise that, for region inference to be tractable, region and effect polymorphism is introduced only for *known functions*, which include functions that are bound directly by a **fun** construct.

Parallelism benefits from region and effect polymorphism, as we have already touched upon in the introduction. The reason is that when a thread calls a function, it can pass private regions to the function, which means that allocation into these regions will not compete for the allocation pointer. In Section 6, we shall discuss suggestions for limiting allocation races further.

Put- and Get-Effects. To decrease the number of regions that are passed to functions, the region type-system and the notion of basic effects are refined to distinguish between basic effects of the form $\mathbf{get}(\rho)$ and $\mathbf{put}(\rho)$. After adjusting each rule to use the refined notion of basic effects, we can refine rule (4) such that the vector of regions $\vec{\rho}'$ parameterised over includes only those regions ρ

that occur on the form $\mathbf{put}(\rho)$ in the effect φ. Now, at application points, the vector of regions passed as arguments must be adjusted so that only regions with put-effects are passed as arguments.

Mutable Objects. The possibility of allocating and accessing mutable objects, such as arrays, is a prerequisite for efficient implementations of many parallel algorithms (some of the benchmarks we present in Section 7 make essential use of mutable data). Region inference supports mutable objects well with the region typing rules for updates and access being straightforward. Both rules introduce an atomic effect $\mathbf{get}(\rho)$, where ρ is the region holding the mutable data structure (no objects are allocated by these operations). Multiple threads can safely access and update different parts of a mutable data-structure, such as an array. Region inference will safely take care of deallocating the mutable objects once the involved threads terminate.

4.4 Region Protection Inference

Based on the notion of put- and get-effects and the notion of region- and effect-polymorphism, we now present a novel effect- and constraint-based algorithm for inferring if a region can be allocated into, simultaneously, by multiple threads.

A *basic constraint* (c) takes the the form $\varphi\#\varphi'$, where φ and φ' are effect sets (containing atomic put-effects and effect variables). A constraint set C is a set of basic constraints. Type schemes are refined to be on the form $\forall\vec{\alpha}\vec{\rho}\vec{\epsilon}.\tau \;/\; C$. A type scheme $\forall\vec{\alpha}\vec{\rho}\vec{\epsilon}.\tau' \;/\; C'$ *generalises* a type τ with constraint set C via $\vec{\rho}'$, written $\sigma \geq \tau/C$ via $\vec{\rho}'$ if there exists $S = (\{\}, S^{\mathrm{t}}, S^{\mathrm{e}})$ such that $S(\tau') = \tau$, $\mathrm{Dom}(S^{\mathrm{t}}) = \vec{\alpha}$, $\mathrm{Dom}(S^{\mathrm{e}}) = \vec{\epsilon}$, and $S(C') = C$. Applying a substitution S to a basic constraint is defined by the rules $S(\{\eta_1, \cdots, \eta_n\}) = \bigcup_i S(\eta_i)$ and $S(\varphi\#\varphi') = S(\varphi)\#S(\varphi')$. The *protection set* of a constraint set C, written $[\![C]\!]$, is defined by the equations $[\![C \cup C']\!] = [\![C]\!] \cup [\![C']\!]$ and $[\![\varphi\#\varphi']\!] = \{\rho | \mathbf{put}(\rho) \in \varphi \cap \varphi'\}$.

Protection inference is defined by a set of rules allowing inferences of the form $\Gamma \vdash e : \sigma, \varphi/C$, which resemble the region typing rules closely but with the addition that a constraint set is inferred and that `letregion` constructs are refined to take the form `letregion` $\rho{:}P$ `in` e, where P takes the form `prot` or `noprot`, signifying whether the region should be protected by a mutex:

Protection Inference $\boxed{\Gamma \vdash e : \sigma, \varphi/C}$

$$\frac{\Gamma \vdash e : \tau, \varphi/C \qquad \varphi'' = \varphi \cup \varphi' \cup \{\rho\} \qquad \Gamma, x : ((\tau, \varphi)T, \rho) \vdash e' : \tau', \varphi'/C' \qquad C'' = C \cup C' \cup \{(\varphi\#\varphi')\}}{\Gamma \vdash \mathtt{letspawn}\ x : ((\tau, \varphi)T, \rho) = e\ \mathtt{at}\ \rho\ \mathtt{in}\ e' : \tau', \varphi''/C''}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau, \varphi/C}{\Gamma \vdash \lambda x : \tau'.e\ \mathtt{at}\ \rho : (\tau' \xrightarrow{\varphi} \tau, \rho), \{\rho\}/C} \qquad \frac{\Gamma \vdash e : (\tau' \xrightarrow{\varphi_0} \tau, \rho), \varphi/C \quad \Gamma \vdash e' : \tau', \varphi'/C'}{\Gamma \vdash e\,e' : \tau, \varphi \cup \varphi' \cup \varphi_0 \cup \{\rho\}/C \cup C'}$$

$$\frac{\Gamma \vdash e : \tau, \varphi/C \quad \rho \notin \mathrm{frv}(\Gamma, \tau) \quad \rho \in [\![C]\!] \Rightarrow P = \mathtt{prot} \quad \rho \notin [\![C]\!] \Rightarrow P = \mathtt{noprot}}{\Gamma \vdash \mathtt{letregion}\ \rho : P\ \mathtt{in}\ e : \tau, \varphi \setminus \{\rho\}/(C \setminus \rho)}$$

$$\frac{\Gamma \vdash e : \tau, \varphi/C \quad \Gamma(f) \geq (\tau \xrightarrow{\epsilon.\varphi_0} \tau', \rho)\ /C' \text{ via } \vec{\rho}}{\Gamma \vdash f\ [\vec{\rho}]\ e : \tau', \varphi \cup \varphi_0 \cup \{\rho\}/C \cup C'} \qquad \ldots$$

Allocation races are introduced only if two simultaneously running threads allocate values in the same region, that is, if two simultaneously executing threads have a non-empty intersection of put-effects. It is perfectly fine that multiple threads read from the same region or that one running thread is allocating into a region and other running threads are reading from the region. In fact, if a thread reads a value from a non-private region, the thread's get-effect on the region will ensure that the region is allocated for as long as the thread is running. This property holds even in a scenario involving mutable objects where, for instance, one thread allocates a value and stores a pointer to the value in a mutable reference cell, which is then dereferenced by another thread. Whereas such a scenario may involve races at the value level, the scenario is entirely safe (from a type-safety

perspective) as long as value updates are atomic. Atomic access to the allocation pointer is required only when simultaneously running threads may allocate into the same region. Notice also that it may be perfectly fine for two different threads to allocate into the same region, as long as the two threads are known not to run simultaneously.

5 IMPLEMENTATION

The implementation that we report on here extends the MLKit Standard ML compiler [Tofte et al. 2022] by adding support in the runtime system to create and join threads, by adjusting the many phases of the compiler to support the new thread constructs, by implementing region protection inference, and by adjusting the code generator to generate instruction sequences that update region allocation pointers atomically. The implementation features two different runtime systems, one based on Pthreads and one preliminary runtime system based on Argobots [Seo et al. 2018], a generic thread-library that supports lightweight user-space scheduling of threads, built on top of Pthreads. In the following we report on the runtime system based directly on Pthreads.

Threads run on the dedicated thread stacks, which, besides from holding stack frames and region descriptors also hold *finite regions*, which are regions that are determined to be allocated into at most once (with values of a statically determined maximum size). Finite regions are implemented unboxed on the stack, which means that a (pointer to a) value in such a region can be communicated to other threads. However, due to the region typing rules, the thread stack holding the value will be live as long as any other thread could potentially access the value.

Although the MLKit makes many high-level optimisations (e.g., specialisation of modules, constant folding, inlining of small functions, tuple elimination, and specialisation of small higher-order recursive functions), it is not an aggressively optimising compiler in the sense that it reorders side-effecting memory operations (e.g., mutable updates), although hardware can still do that. Whereas synchronisation of shared mutable data is outside the scope of this paper, recent research suggests [Vollmer et al. 2017] that inserting a memory fence after every shared mutable update has low overhead for functional programs where such updates are (relatively) rare. Such a strategy provides sequential consistency and could be further optimised, using a variation of protection inference, by avoiding fences when updating objects in private regions.

5.1 Thread Local Data

In the non-parallel version of the MLKit, the runtime system maintains a number of global variables including the `REGION_TOP` variable, which holds a pointer to the top-most region, and the `FREE_LIST` variable, which holds the application's list of free region pages, pages that are allocated, but have been freed by the region runtime system. In the parallel versions of the runtime system, each thread has its own `REGION_TOP` value, which is accessed through a dedicated *thread-context* register, which give immediate access to thread-local data, including thread-local free lists of region pages.

For the Pthreads runtime system, when a thread frees a page, the page is added to the thread's free list. Only when a thread needs a new region page and no page is available in the thread-local free list, the main free list is accessed and a number of pages are moved from the shared free-list to the thread-local free-list (and the mutex that protects the main free list is temporarily locked). When a thread terminates, thread-local pages are added to the main free list. Without this intermediate free-list layer, it turns out that allocations in threads will too often be involved in context switches that are caused by locking and unlocking the mutex that protects the main free list.

5.2 Exceptions

A thread's top-most exception handler writes the exception value into the thread's local data. When a thread value is retrieved by the `get` function, the code joins the thread and waits for the thread to

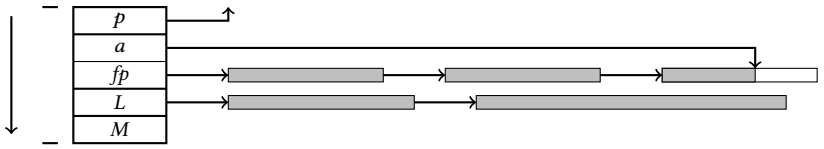


Fig. 5. A region descriptor on the down-growing stack.

terminate. The code then tests if an exception value is stored in the thread's local data, in which case, the exception is raised by the parent. If no exception is stored, the calculated value is returned.

5.3 Atomic Allocation

In general, region allocation races can occur if two different threads allocate into the same region. In the following we shall consider only the case of allocations into *non-bounded regions* (as opposed to finite regions, which we know statically are allocated into at most once).

A *region descriptor*, which is depicted in Figure 5, represents an unbounded region and consists of a pointer (*fp*) to the previous region descriptor on the stack (possibly NULL), a pointer (*fp*) to a list of 8KiB-aligned fixed-sized region pages, each of size 8KiB, a pointer (a) to the first free location in the last region page (i.e., the *allocation pointer*), a pointer (L) to a linked list of `malloc`-allocated large objects, and a mutex (M) for resolving allocation races (NULL if no protection is needed).

Space for region descriptors are reserved as part of a function's stack frame. When a region is allocated, the associated region descriptor is populated with a pointer to a fresh region page, taken from the threads' free-list (possibly by taking a number of pages from the process's `FREE_LIST`), and the allocation pointer is initialised to point at the start of the region page. Moreover, the "previous pointer" in the region descriptor is initialised to point at the thread's current content of the thread's `TOP_REGION` value, after which the thread's `TOP_REGION` value is updated to point to the new region descriptor. Finally, the large-object pointer is initialised to NULL, and possibly, a mutex is installed from a free list of mutexes (managed the same way as region pages).

For allocating in a region, the runtime system implements a fallback allocation routine `alloc`, which takes as arguments a region r (i.e., a pointer to a region descriptor) and a number n, which indicates the number of words to allocate. The allocation routine uses Compare-And-Swap (CAS) operations and the region's mutex to protect against allocation races. Space is allocated in the large object list if n is larger than the size of a region page. Otherwise, if there is space in the current region page (detected using region-page alignment-properties), $r.a$ is updated to $r.a + n$ and the previous content of $r.a$ is returned. In case a new page is needed, a page is fetched from the thread's free-list; if this list is empty, first, a number of pages are fetched from the global free-list. If this free list is also empty, a number of new pages are allocated using `malloc`.

For allocating n words in a region r, the generated assembler code attempts to avoid calling `alloc` (for avoiding using the mutex) by applying the following steps:

(1) If n is larger than the page size (8KiB), return the result of calling the `alloc` routine.
(2) Save the current value of $r.a$ in a temporary register R.
(3) Using the region page alignment properties, detect if $R+n$ is within the region page boundary; if not, attempt to update (using a CAS operation with R) a to point to the end of the page; on success, return the result of calling the `alloc` routine; on failure, jump to step (2).
(4) Use a CAS operation with R to update $r.a$ to contain the value $R + n$. On success, return the content of R. On failure, jump to step (2).

Notice that before the `alloc` routine is taking the region's mutex lock, the routine takes care of using a CAS operation to store the region page boundary in the allocation pointer $r.a$. In this way, it is known that the allocation inline-instruction-sequence will not concurrently update $r.a$.

The above allocation strategy allows for multiple threads to store into the same region and is in many cases a good compromise between safety and performance, in particular, when combined with region protection inference.

Much like MPL expects programs to be written to ensure that there is no entanglement [Westrick et al. 2019], MLKit programmers should avoid that different threads allocate local data in the same shared region, which can sometimes be ensured by simple copying of values into private regions.

In the MLKit, protection inference runs after region inference as a separate pass that generates constraints for each subexpression as defined by the rules in Section 4.4. Whenever a **`letregion`** construct is visited, it is decided, based on the constraints generated for the local expression, whether the bound region should be protected (i.e., have a mutex in the region descriptor) or not.

6 ALTERNATIVES TO AVOIDING REGION ALLOCATION RACES

In general, allocation races may occur if two simultaneously running threads allocate into the same region. We make the following observations:

(1) Regions that are allocated into only by a single thread (e.g., the thread that allocates the region) need no allocation protection. When such regions are allocated into, no allocation races arise (unless a new page is needed from the global free-list of pages).
(2) Reading (values) from a region (no matter whether it is local or not) requires no locking as only values created before the thread was spawned are accessible. Values that are assigned to mutable references or arrays are valid values once assigned and cause no allocation races.

There are various ways in which such data races can be eliminated. One possibility is to protect allocation in a region using a combination of CAS operations and a mutex, as discussed in the previous section, which is the current generic approach taken in the MLKit implementation.

Shadow Regions. To better support scalable allocation (where many threads allocate into the same regions), an alternative to the atomic allocation scheme of Section 5.3 is for each thread to maintain its own local "shadow regions" for regions that the thread allocates into and that are potentially also allocated into by other threads. Upon exiting the scope of a thread, the thread's shadow regions can then be merged with the parent thread's regions by linking region pages (this operation can be done in constant time, but may introduce region fragmentation). With the current implementation, however, it will be difficult, with little overhead, to implement the required region descriptor mappings. This idea is similar to how traditional parallel runtime systems may allow each thread to pre-allocate a block of memory for local use, with the benefit of reducing allocation congestion. With region memory-management, however, different allocations may store into different regions, which complicates block management and perhaps suffers from fragmentation. We consider these design opportunities possibilities for future work.

Higher-Order Region Polymorphism. Another possibility for reducing allocation races would be to improve the precision of region inference, for instance by supporting higher-order region-polymorphism. Consider an implementation of a parallel prefix scan on an array of values:

```
val scan : (α * α → α) → α → α array → α array
```

To implement this routine in parallel, the programmer may choose to split the array in two, scan the two parts in parallel (using the supplied associative operator and neutral element), and use a data-parallel operation (e.g., a map) to update each element in the second array using the last element of the first result array. From the perspective of the `scan` function, the region type for

the passed associative function f is $(\alpha * \alpha, \rho) \xrightarrow{\epsilon.\emptyset} \alpha$. Here `scan` itself will need to allocate the argument pairs to f. Because f is inherently region monomorphic, the pairs will be allocated in the same region. What is needed is for f to be region polymorphic in the region holding the pairs:

```
val scan : (∀ρ.(α * α, ρ) → α) → α → α arr → α arr
```

Now, the two threads used for scanning the left and right parts of the argument array can use different regions for storing the pairs. However, consider if α is instantiated with a boxed type such as $(\texttt{real}, \rho)$, in which case f would both allocate in ρ and read from ρ. In this case, a more elaborate combination of higher-order type- and region-polymorphism would be necessary. Currently, region inference, as it is implemented in the MLKit, does not support higher-order region-polymorphism, a restriction that potentially could be lifted by supporting explicit region annotations.

7 EXPERIMENTAL RESULTS

In this section, we present a series of experiments that serve to demonstrate the viability of using region inference as the primary memory management discipline for parallel execution of high-level functional programs on multi-core architectures. We show parallel scalability for MLKit and MPL on a collection of benchmark programs, compared to sequential baselines. The goal here is to show that parallel MLKit shows reasonable scalability, compared to a state-of-the-art implementation.

7.1 Experimental Setup and Benchmark Overview

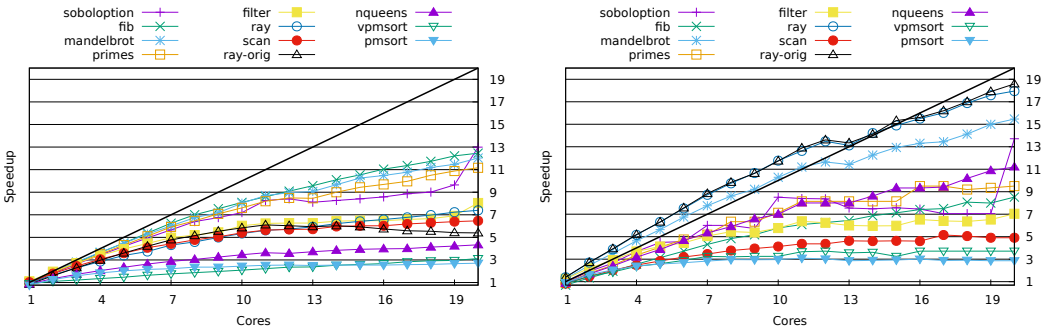
The experiments are performed with MLKit version 4.7.3, MPL MPL version 0.3, and MLton 20210117. MPL is built on MLton, which is a whole-program highly-optimising compiler, while MLKit features a smart-recompilation system that allows for quick rebuilds upon modification of source code [Elsman 2008]. In particular, MPL's optimiser uses aggressive unboxing and defunctionalisation, sometimes leading to significantly faster sequential code (see Table 1). We generally do not expect MLKit to match MPL in absolute performance, and our focus here is on parallel scalability. In principle, there is nothing that prevents a compiler from using both region inference and aggressive MLton-style optimisations, but doing so is beyond the scope of this paper. When reporting speedups, we are comparing parallel MLKit with sequential MLKit, and parallel MPL with sequential MLton.

All benchmark programs are executed on an Intel Xeon Gold 6230 CPU, with 192GiB of RAM, 20 cores, and hyper-threading disabled. The MLKit benchmarks are compiled with reference-tracing garbage collection disabled, meaning they rely entirely on region inference. Recent work [Elsman and Hallenberg 2021] compares the performance of a pure region-memory management scheme, based on region inference, and a scheme that combines region inference and garbage collection.

The MPL programs are informed how many cores are available (and hence how many OS threads to spawn). The MLKit benchmark implementations are oblivious and always use approximately 50 threads independent of the numbers of cores available, and have the number of cores they can access limited at the OS level via `taskset`. This difference is because MPL contains its own scheduler for scheduling "lightweight" threads across a fixed number of "heavy" OS threads, while MLKit's threading interface always launches a new OS thread, and leaves granularity control to user code. Benchmark code is shared between all compiler instances and make use of a simple parallel programming module (essentially, the interface shown in Section 2) that has distinct implementations for MPL, MLKit, and sequential Standard ML.

We measure the average time of 10 runs within the same process. Peak memory usage is measured with `time -l`, which is therefore the total for all 10 runs.

The ***Soboloption*** benchmark prices an equity call option via the Black-Scholes formula [Black and Scholes 1973] implemented with quasi-random Sobol numbers [Bratley and Fox 1988]. In high-level terms, this is a `map-reduce` composition. ***Fib*** computes $fib(46)$ with the classic (inefficient)



(a) MLKit speedup over sequential MLKit.

(b) MPL speedup over sequential MLton.

Fig. 6. Scalability of parallel MLKit and MPL measured from 1 to 20 cores.

recursive formula, parallelising the recursive calls three levels deep, then switching to sequential execution. ***Mandelbrot*** is a visualisation of the Mandelbrot set, using an iteration limit of 10^5 and a radius of 2^8. This means there is substantial sequential work per pixel. ***Primes*** uses parallel `map` and `filter` to find all prime numbers below 10^6. ***Filter*** is a parallel filter that finds all integers in the range $[0, 10^7]$ that are divisible by 99. ***Ray*** is a BVH-accelerated ray tracer adapted from MPL's benchmark suite. In its original formulation, it represents 3D vectors with SML records of reals, which are boxed in MLKit. We have modified the code to use an unboxed representation instead. ***Scan*** computes a parallel prefix sum of the integers in the range $[0, 10^8]$. ***Ray-orig*** is the *ray* benchmark without the modification mentioned above. This version runs significantly slower in MLKit, but not in MPL, as MPL, in this case, automatically unboxes the reals in the records. ***Nqueens*** solves the classic N-queens problem for $N = 13$, using a recursive search. ***Pmsort*** is parallel merge sort on lists with 10^6 elements, as presented in Section 2. ***Vpmsort*** is similar to *pmsort*, but on arrays instead of lists. Some of the benchmark programs (e.g., *mandelbrot*, *ray*, *scan*, and *ray-orig*) use shared mutable arrays while others (e.g., *vpmsort*, *pmsort*, and *nqueens*) are purely functional. The benchmarks *fib*, *ray-orig*, and *nqueens* are from the MPL benchmark suite.

7.2 Benchmark Results

Figure 6 shows scalability of MLKit and MPL when varying the number of available cores. Table 2 shows memory usage for the sequential baseline programs, and the proportional increase in memory usage when executing in parallel on 20 cores. Table 1 shows the absolute runtimes of the sequential baseline programs ($\textbf{MLKit}_{\text{seq}}$, $\textbf{MLton}_{\text{seq}}$), the parallel programs running on a single thread ($\textbf{MLKit}_1$, $\textbf{MPL}_1$), and the parallel programs running with 20 threads ($\textbf{MLKit}_{20}$, $\textbf{MPL}_{20}$).

Speedups. As shown on Figure 6a, the program that scales the best with MLKit is *fib*, likely because it performs no heap allocations within each thread. This benchmark thus represents the current "speed limit" of parallel MLKit. *Soboloption*, *mandelbrot*, *primes*, *filter*, *scan*, and *ray* all scale decently, and represent regular parallelism, although with some irregular workload. This is handled by *over-partitioning*, as we launch more threads than the hardware is physically capable of executing. The slightly worse scaling of *ray* is because it is more memory-bound, as each pixel requires us to traverse an irregular tree-based data structure. The benchmarks that scale the poorest (*nqueens*, *ray-orig*, *vpmsort*, and *pmsort*) suffer slowdown simply from being compiled with MLKit's parallel code generator, as shown by the runtimes in Table 1. The reason for this slowdown is that, compared to the sequential code generator, the parallel code generator uses a more complex

Table 1. Absolute benchmark runtimes.

Benchmark	$\text{MLKit}_{\text{seq}}$	MLKit_{1}	MLKit_{20}	$\text{MLton}_{\text{seq}}$	MPL_{1}	MPL_{20}
soboloption	3.820*s*	4.090*s*	0.294*s*	2.468*s*	2.580*s*	0.180*s*
fib	10.104*s*	10.365*s*	0.811*s*	12.875*s*	18.652*s*	1.511*s*
mandelbrot	10.428*s*	10.461*s*	0.867*s*	10.285*s*	8.195*s*	0.665*s*
primes	2.132*s*	2.161*s*	0.191*s*	0.570*s*	0.634*s*	0.060*s*
filter	5.077*s*	4.565*s*	0.631*s*	2.682*s*	2.134*s*	0.381*s*
ray	2.574*s*	2.725*s*	0.348*s*	3.411*s*	2.456*s*	0.190*s*
scan	1.151*s*	1.111*s*	0.178*s*	0.829*s*	1.010*s*	0.169*s*
ray-orig	9.687*s*	11.468*s*	1.796*s*	3.402*s*	2.435*s*	0.183*s*
nqueens	1.506*s*	1.708*s*	0.347*s*	1.120*s*	1.305*s*	0.100*s*
vpmsort	0.281*s*	0.314*s*	0.090*s*	0.260*s*	0.320*s*	0.070*s*
pmsort	0.331*s*	0.431*s*	0.122*s*	0.770*s*	0.841*s*	0.269*s*

instruction sequence (involving conditional synchronisation) for allocations into regions that may be stored into by multiple threads (see Section 5.3). Scaling compared to parallel MLKit on 1 core is decent for these benchmarks, as this constant overhead is independent of the number of threads.

A detailed analysis of MPL's performance on these benchmarks is outside our scope. We note that Figure 6b shows that *pmsort* scales as poorly with MPL as MLKit. While *ray-orig* scales excellently with MPL due to its effective unboxing optimisations, *scan* and *filter* do not perform very well. Also notice that MPL with one core often performs better than sequential MLton, which we speculate is due to optimisations that are implemented in MPL but not in MLton. This aspect also explains some of the high scalability numbers (e.g., for *mandelbrot*, *ray*, and *ray-orig*) as MPL speedup is calculated relative to sequential MLton.

Impact of Protection Inference. As a separate experiment, we have measured the performance impact of protection inference. The slowdown from making all allocations atomic ranges from 0 on *fib* and *primes*, to 30× on *pmsort*, with a geomean slowdown of 1.87×.

Space Usage. Table 2 shows the increase in memory usage when executing the benchmark programs in parallel. Generally, the increase is modest. This is because many programs consist of threads mutating a shared array, where the memory usage is largely independent of the number of threads. The common outliers are *ray*, *fib*, and *nqueens*, where particularly the latter two use very little memory in sequential execution. Merely allocating additional stacks for parallel execution becomes the main cost. Some benchmarks (*ray*, *ray-orig*, *vpmsort*, and *pmsort*) use substantially less memory when compiled with MPL (or MLKit) as compared to MLton, which may simply be due to differences in MLton's and MPL's garbage collectors.

8 RELATED WORK

A key strand of related work is the previous work on adding support for parallel OS threads in ML-like languages, including OCaml and Standard ML, which both feature mutable data structures such as arrays and references. Cooper and Morrisett describe, in [Cooper and Morrisett 1990], several implementations of a Standard ML thread interface, building on top of SML/NJ, of which one of their implementations executes on native OS threads by separating the heaps evenly among threads and by using a stop-the-world garbage collection strategy. Newer approaches to adding parallelism to ML-like languages are divided into shared-memory approaches and message-passing approaches. The first kind of approach focuses on retrofitting OS-level thread support into highly-performing sequential language implementations, such as the native OCaml compiler [Sivaramakrishnan et al. 2020] and the MLton Standard ML compiler [Westrick et al. 2019], which, in both cases, require

Table 2. The increase in memory usage for parallel execution with MLKit and MPL, compared to their respective sequential baselines.

Benchmark	$\text{MLKit}_{\text{seq}}$	MLKit_{1}	MLKit_{20}	$\text{MLton}_{\text{seq}}$	MPL_{1}	MPL_{20}
soboloption	1MiB	3.19×	3.26×	1MiB	96.28×	29.96×
fib	1MiB	5.60×	5.60×	768KiB	1.71×	7.12×
mandelbrot	7MiB	2.42×	2.46×	7MiB	1.03×	1.56×
primes	40MiB	1.27×	1.27×	20MiB	0.84×	1.09×
filter	3060MiB	1.00×	1.00×	1963MiB	0.98×	0.98×
ray	26MiB	4.85×	4.87×	151MiB	0.11×	0.47×
scan	764MiB	1.00×	1.00×	811MiB	0.94×	0.95×
ray-orig	170MiB	4.03×	4.08×	151MiB	0.11×	0.44×
nqueens	2MiB	68.74×	116.92×	1MiB	15.77×	62.82×
vpmsort	36MiB	2.02×	2.05×	123MiB	0.40×	0.79×
pmsort	77MiB	1.89×	1.97×	738MiB	0.28×	0.75×

special attention to the underlying garbage collection techniques, in particular with respect to dealing efficiently with mutable effects. In the case of MPL, which adds shared-memory Fork-Join parallelism to the MLton Standard ML compiler through a simple `par`-function, the runtime overhead of non-parallel executables was initially reported to be roughly 200 percent, which is a significant overhead to pay for the potential to run code in parallel [Raghunathan et al. 2016]. The MPL semantics is based on the notion of disentangled heaps, which is a memory discipline where each thread is associated with an individual heap and where pointers between heaps can only point upwards towards the root [Raghunathan et al. 2016]. For programs that guarantee that heaps are always disentangled, independent garbage collection of leaf heaps is supported with provably good space-efficient properties [Arora et al. 2021]. Recent work improves performance significantly and support has been added for mutable objects to be assigned to as long as the mutable object appears in an ancestor's heap [Westrick et al. 2019]. The disentangled-heap property, which is required by MPL, is enforced automatically by a combination of static and dynamic techniques [Westrick et al. 2022]. If a thread assigns to an object allocated by a sibling, the object will be allocated in a memory region allocated by a common ancestor.

Another parallel ML-like language is Manticore [Fluet et al. 2008], which supports both light-weight threads and native Pthreads-like threads for providing a range of parallel programming techniques, including Fork-Join style parallel programming, CML-style programming [Reppy et al. 2009], and data-parallel programming based on parallel reductions, parallel scans, and parallel maps. Manticore can be configured to work with different backends and stack representations [Farvardin and Reppy 2020], but all are based on reference-tracing garbage collection techniques.

Lightweight threads, based on message-passing techniques, as supported by Go [Davis et al. 2012] and Erlang [Armstrong 2003] have also been added to Standard ML in the context of MultiMLton [Li et al. 2016; Sivaramakrishnan et al. 2010, 2014]. Such lightweight message passing frameworks provide a different model for parallelism than the shared-memory Fork-Join style model. In the work we present here, we base all scheduling on the underlying Pthreads implementation or, alternatively, on the Argobots lightweight-thread implementation [Seo et al. 2018]. We consider it future work to investigate techniques for properly supporting lightweight user-space threads, perhaps based on the MassiveThreads library [Nakashima and Taura 2014], which serves as the basis for prototype thread support in SML# [Ohori et al. 2017].

Also related to this work is the previous work on combining region inference and garbage collection in the MLKit [Elsman 2023; Hallenberg et al. 2002] and how the concept of typed regions can be used to support untagged values and a combination of region inference and generational

garbage collection [Elsman and Hallenberg 2021]. The implementation that we describe in the present paper does not support reference tracing garbage collection, although we believe the implementation can be extended to support parallel collection of regions that live in leaf threads.

Cyclone [Swamy et al. 2006] is a region-based type-safe C dialect, for which, the programmer can decide if an object should reside in the GC heap or in a region. Cyclone interfaces with Pthreads [Gerakios et al. 2010]. Another region-based language is Gay and Aiken's RC system, which features limited explicit regions for C, combined with reference counting of regions [Gay and Aiken 2001]. Whereas our implementation has limited support for explicit region annotations, it is primarily based on the assumption that region annotations are inferred by the compiler. Also related to the present work is the work by Aiken et al. [Aiken et al. 1995], who show how region inference may be improved for some programs by removing the constraints of the stack discipline. Other approaches for improving region inference, includes techniques for deallocating parts of a region, which has been investigated in the context of Go [Davis et al. 2013]. Finally, another body of related work is on using region inference without combining it with a reference-tracing garbage collector. Such work include the use of region inference as the primary memory management scheme for a web server [Elsman and Hallenberg 2003; Elsman and Larsen 2004; Elsman et al. 2018].

A modern language for system programming is Rust, which is based on ownership types for managing resources, including memory [Aldrich et al. 2002]. Ownership types are also used for real-time implementations of Java [Boyapati et al. 2003]. Compared to these techniques, region inference provides an automatic technique for inferring allocation and deallocation of memory.

From a parallelism perspective, there is a large body of related work on supporting deterministic parallelism in Haskell, for instance through the use of the Par-monad [Marlow et al. 2011] and through software transactional memory (STM) [Harris et al. 2005; Shavit and Touitou 1995]. Another strand of work on functional parallel programming include the work on embedded data-parallel languages, such as Accelerate [Chakravarty et al. 2011] and functional data-parallel languages, such as Futhark [Henriksen et al. 2017] and SaC [Grelck and Scholz 2007]. These languages are centered around the array programming paradigm and are superior when it comes to targeting massively parallel architectures, such as GPUs, but are not necessarily useful for task-parallelism.

Other work investigates parallel garbage collection techniques to decrease garbage collection times [Marlow et al. 2008]. Our present implementation of parallelism support in the MLKit compiler does not support reference-tracing garbage collection and we consider it future work to investigate the possibility of leaf threads to perform garbage collection of local regions in parallel.

9 CONCLUSION AND FUTURE WORK

We have presented a framework that integrates region inference and parallel programming with threads and shown that region inference can be a helpful companion for managing memory in a parallel setting. We have implemented the techniques and demonstrated the performance of the approach based on a set of non-trivial benchmarks. We have compared the performance with versions compiled with MPL. We have demonstrated that the techniques are promising and that our implementation in some cases performs as well as MPL.

There are a number of opportunities for future work, including adding support for parallel reference-tracing garbage collection of regions that are private to a leaf heap. Another candidate for future work is to improve the scalability properties of the approach, in particular by supporting lightweight user-space thread scheduling and by providing techniques for localising region allocation, and thereby better support scalable allocation. Another candidate for future work is to support region-explicit annotations in source programs, in particular with support for specifying effect properties of functions and individual threads.

DATA AVAILABILITY STATEMENT

An artifact demonstrating the benchmark results presented in the paper is archived in Zenodo [Elsman and Henriksen 2023].

REFERENCES

Alexander Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 174–185. https://doi.org/10.1145/207110.207137

Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. 2002. Alias Annotations for Program Understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, USA) *(OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 311–330. https://doi.org/10.1145/582419.582448

Joe Armstrong. 2003. *Making Reliable Distributed Systems in the Presence of Software Errors.* Ph. D. Dissertation. The Royal Institute of Technology, Stockholm, Sweden.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space-Efficient Parallel Functional Programming. *Proc. ACM Program. Lang.* 5, POPL, Article 18 (jan 2021), 33 pages. https://doi.org/10.1145/3434299

Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From Region Inference to von Neumann Machines via Region Representation Inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 171–183. https://doi.org/10.1145/237721.237771

Fischer Black and Myron Scholes. 1973. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy* 81, 3 (1973), 637–654. http://www.jstor.org/stable/1831029

Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. 2003. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 324–337. https://doi.org/10.1145/781131.781168

Paul Bratley and Bennett L. Fox. 1988. Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator. *ACM Trans. Math. Softw.* 14, 1 (mar 1988), 88–100. https://doi.org/10.1145/42288.214372

Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

E. Cooper and G. Morrisett. 1990. *Adding Threads to Standard ML.* Technical Report CMU-CS-90-186. Carnegie Mellon University, Department of Computer Science. Technical report.

Matthew Davis, Peter Schachte, Zoltan Somogyi, and Harald Søndergaard. 2013. A Low Overhead Method for Recovering Unused Memory inside Regions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (Seattle, Washington) *(MSPC '13)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2492408.2492415

Matthew Davis, Peter Schachte, Zoltan Somogyi, and Harald Søndergaard. 2012. Towards region-based memory management for Go. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness.* 58–67. https://doi.org/10.1145/2247684.2247695

Martin Elsman. 2003. Garbage Collection Safety for Region-Based Memory Management. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New Orleans, Louisiana, USA) *(TLDI '03)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/604174.604190

Martin Elsman. 2008. *A Framework for Cut-Off Incremental Recompilation and Inter-Module Optimization.* Technical Report. IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark.

Martin Elsman. 2023. Garbage-Collection Safety for Region-Based Type-Polymorphic Programs. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI 2023)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3591229

Martin Elsman and Niels Hallenberg. 1995. An Optimizing Backend for the ML Kit Using a Stack of Regions. Student Project 95-7-8, University of Copenhagen (DIKU).

Martin Elsman and Niels Hallenberg. 2003. Web Programming with SMLserver. In *International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag, 74–91. https://doi.org/10.1007/3-540-36388-2_7

Martin Elsman and Niels Hallenberg. 2021. Integrating region memory management and tag-free generational garbage collection. *Journal of Functional Programming* 31 (2021), e4. https://doi.org/10.1017/S0956796821000010

Martin Elsman and Troels Henriksen. 2023. Artifact for the PLDI 2023 paper: Parallelism in a Region Inference Context. Zenodo. https://doi.org/10.5281/zenodo.7810545

Martin Elsman and Ken Friis Larsen. 2004. Typing XHTML Web Applications in ML. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*. Springer-Verlag. https://doi.org/10.1007/978-3-540-24836-1_16

Martin Elsman, Philip Munksgaard, and Ken Friis Larsen. 2018. Experience Report: Type-Safe Multi-Tier Programming with Standard ML Modules. In *Proceedings of the ML Family Workshop* (St. Louis, Missouri, USA) *(ML '18)*.

Kavon Farvardin and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 75–90. https://doi.org/10.1145/3385412.3385994

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2008. Implicitly-Threaded Parallelism in Manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/1411204.1411224

David Gay and Alex Aiken. 2001. Language Support for Regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/378795.378815

Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2010. Race-Free and Memory-Safe Multithreading: Design and Implementation in Cyclone. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/1708016.1708020

Clemens Grelck and Sven-Bodo Scholz. 2007. SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming* (Nice, France) *(DAMP '07)*. Association for Computing Machinery, New York, NY, USA, 25–33. https://doi.org/10.1145/1248648.1248654

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/512529.512563

Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/512529.512547

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

Muyuan Li, Daniel E. McArdle, Jeffrey C. Murphy, Bhargav Shivkumar, and Lukasz Ziarek. 2016. Adding Real-Time Capabilities to a SML Compiler. *SIGBED Rev.* 13, 2 (April 2016), 8–13. https://doi.org/10.1145/2930957.2930958

Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. 2008. Parallel Generational-Copying Garbage Collection with a Block-Structured Heap. In *Proceedings of the 7th International Symposium on Memory Management* (Tucson, AZ, USA) *(ISMM '08)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/1375634.1375637

Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell* (Tokyo, Japan) *(Haskell '11)*. ACM, New York, NY, USA, 71–82. https://doi.org/10.1145/2034675.2034685

Greg Morrisett. 1995. *Compiling with Types*. Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Jun Nakashima and Kenjiro Taura. 2014. *MassiveThreads: A Thread Library for High Productivity Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238. https://doi.org/10.1007/978-3-662-44471-9_10

M. H. A. Newman. 1942. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics* 43, 2 (1942), 223–243. http://www.jstor.org/stable/1968867

Atsushi Ohori, Kenjiro Taura, and Katsuhiro Ueno. 2017. Making SML# a General-purpose High-performance Language. In *Draft Proceedings of the ML family workshop (ML '2017)*.

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 392–406. https://doi.org/10.1145/2951913.2951935

John Reppy, Claudio V. Russo, and Yingqi Xiao. 2009. Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/1596550.1596588

Guillaume Salagnac, Chaker Nakhli, Christophe Rippert, and Sergio Yovine. 2006. Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*.

Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2018), 512–526. https://doi.org/10.1109/TPDS.2017.2766062

Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottowa, Ontario, Canada) *(PODC '95)*. Association for Computing Machinery, New York, NY, USA, 204–213. https://doi.org/10.1145/224964.224987

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113 (Aug. 2020), 30 pages. https://doi.org/10.1145/3408995

KC Sivaramakrishnan, Lukasz Ziarek, Raghavendra Prasad, and Suresh Jagannathan. 2010. Lightweight Asynchrony Using Parasitic Threads. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming* (Madrid, Spain) *(DAMP '10)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/1708046.1708059

K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for Standard ML. *Journal of Functional Programming* 24, 6 (2014), 613–674. https://doi.org/10.1017/S0956796814000161

Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe manual memory management in Cyclone. *Science of Computer Programming* 62, 2 (2006), 122–144. https://doi.org/10.1016/j.scico.2006.02.003 Special Issue: Five perspectives on modern memory management - Systems, hardware and theory.

Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (jul 1998), 724–767. https://doi.org/10.1145/291891.291894

Mads Tofte and Lars Birkedal. 2000. Unification and Polymorphism in Region Inference. *Proof, Language, and Interaction. Essays in Honour of Robin Milner* (May 2000). (25 pages).

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (01 Sep 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. 2022. *Programming with Regions in the MLKit (Revised for Version 4.7.2)*. Technical Report. IT University of Copenhagen, Denmark.

Michael Vollmer, Ryan G. Scott, Madanlal Musuvathi, and Ryan R. Newton. 2017. SC-Haskell: Sequential Consistency in Languages That Minimize Mutable Shared Heap. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 283–298. https://doi.org/10.1145/3018743.3018746

Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1159876.1159877

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection with Near-Zero Cost. *Proc. ACM Program. Lang.* 6, ICFP, Article 115 (aug 2022), 32 pages. https://doi.org/10.1145/3547646

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2019. Disentanglement in Nested-Parallel Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 47 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371115

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (nov 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Received 2022-11-10; accepted 2023-03-31