JFP **31**, e4, 31 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S0956796821000010

Integrating region memory management and tag-free generational garbage collection

University of Copenhagen, Denmark
(*e-mail:* `mael@di.ku.dk`)

NIELS HALLENBERG

SimCorp A/S, Denmark
(*e-mail:* `niels.hallenberg@simcorp.com`)

Abstract

We present a region-based memory management scheme with support for generational garbage collection. The scheme features a compile-time region inference algorithm, which associates values with logical regions, and builds on a region type system that deploys region types at runtime to avoid the overhead of write barriers and to support partly tag-free garbage collection. The scheme is implemented in the MLKit Standard ML compiler, which generates native x64 machine code. Besides demonstrating a number of important formal properties of the scheme, we measure the scheme's characteristics, for a number of benchmarks, and compare the performance of the generated executables with the performance of executables generated with the MLton state-of-the-art Standard ML compiler and configurations of the MLKit with and without region inference and generational garbage collection enabled. Although region inference often serves the purpose of generations, combining region inference with generational garbage collection is shown often to be superior to combining region inference with non-generational collection despite the overhead introduced by a larger amount of memory waste, due to region fragmentation.

1 Introduction

Region-based memory management allows programmers to associate lifetimes of objects with so-called regions and to reason about how and when such regions are allocated and deallocated. Region-based memory management, as it is implemented for instance in Rust (Aldrich *et al.*, 2002), can be a valuable tool for constructing certain kinds of critical systems, such as real-time embedded systems (Salagnac *et al.*, 2006). Region inference differs from explicit region-based memory management by taking a non-annotated program as input and producing 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.*, 2006)) to obtain good space and time performance for critical parts of the program.

The region-based memory management 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', 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` term 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', and finally, deallocate the region bound to ρ (and its pages). The region type system allows regions to be passed to functions at run time (i.e., functions can be region-polymorphic) and to be captured in closures. The soundness of region inference ensures that a region is not deallocated as long as a value within it may be used by the remainder of the computation.

To remedy the problem that region inference does not always capture precisely the lifetime properties of objects, previous work has augmented the static inference scheme with a more dynamic lifetime-based reference-tracing copying garbage collector (Hallenberg *et al.*, 2002). For such an integration of region-based memory management and reference-tracing garbage collection, care must be taken to rule out the possibility of deallocating regions with incoming pointers from live objects. Incidentally, it turns out that such pointers can be ruled out by the region type system (Elsman, 2003), which means that we can be sure that a tracing garbage collector will not be chasing dangling pointers at run time.

The resulting combined scheme works well in practice and forms the basis of the MLKit, a bootstrapping complete implementation of the Standard ML language. Although it turns out that region inference, for a variety of benchmarks, drastically decreases the time spent on reference tracing garbage collection, the lack of generational garbage collection causes all live values to be visited for every garbage collection. In essence, both region inference and generational garbage collection have been shown to manage short-lived values well. It has not been clear, however, whether the two approaches could complement each other. In this paper, we present a framework that combines region inference and generational garbage collection, and discuss the effects of the integration. The generational collector associates two generations with each region. It has the feature that an object is promoted to the old generation of its region (during a collection) only if it has survived a previous collection. Compared to the earlier non-generational collection technique (Hallenberg *et al.*, 2002), we may run a minor collection by only traversing (and copying) objects in the young generations.

The traditional implementations of reference tracing garbage collection lead to a representation of values that allows for dynamically detecting the structural type of a value. Based on a notion of region types, we demonstrate that the combination of region inference and generational garbage collection allows for tag-free compact memory representations for a variety of common data structures, such as pairs, lists, and trees.

The contributions of this paper are the following:

1. We present a technique for combining region-based memory management with a generational (stop the world) garbage collector, using a notion of typed regions, which allows us to deal with mutable data in minor collections and for tag-free representations of certain kinds of values, such as tuples.
2. We present a region type system that guarantees that regions contain values according to the regions' types. The region type system refines an earlier region type system that guarantees that no dangling pointers are introduced during evaluation (Elsman, 2003).

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

3. To demonstrate the feasibility of the technique, we show empirically that the MLKit generates code that, in many cases, is comparable in performance to executables generated with the MLton compiler (v20201023).
4. We demonstrate empirically, based on a large set of benchmarks, that combining region inference with generational garbage collection is often superior to combining region inference with non-generational garbage collection despite the overhead introduced by a larger amount of memory waste, due to region fragmentation.

The study is performed in the context of the MLKit (Tofte *et al.*, 2006). It generates native x64 machine code for Linux and macOS (Elsman & Hallenberg, 1995) and implements a number of techniques for refining the representations of regions (Birkedal *et al.*, 1996; Tofte *et al.*, 2004), including dividing regions into stack allocated (bounded) regions (also called finite regions) and heap allocated regions.

The paper is organised as follows. In the following section, we first give an informal example demonstrating how a Standard ML implementation of Mergesort on lists is compiled and represented in the region-based target language and how the program can be optimised by a region-aware programmer to make better use of memory. While the example serves as an introduction to region-based memory management in the MLKit, it also serves to demonstrate that, without help from a region-aware programmer, the program will benefit from using a combination of region inference and generational garbage collection. In Section 3, we present a simplified, but formal, region type system for a language that serves as a target language for region inference. We present a number of properties of the type system, including region type soundness and the property that no dangling pointers are introduced during evaluation. In Section 4, we present the generational garbage collection algorithm and show how the algorithm is extended to work with mutable and large objects. In Section 5, we present a number of experimental results and evaluate the work. In Section 6, we describe related work, and in Section 7, we conclude.

2 An introduction to region-based memory management

Mergesort on lists can be implemented elegantly using pattern-matching in a functional language such as Haskell or ML. Figure 1 lists a Standard ML implementation of the Mergesort algorithm. The implementation consists of a function `split`, which splits a list in two, a function `merge`, which merges two sorted integer lists, and a function `msort`, which makes use of `split` and `merge` for sorting the input.

Region inference will associate region types to the expressions in the program. Moreover, based on the inferred types, it may further infer that a function takes regions as arguments and that other regions can be allocated temporarily within a function. The `merge` function is inferred to have the following type:[1]

$$\texttt{val merge : } \forall\rho.\ \texttt{(int,}\rho\texttt{)list * (int,}\rho\texttt{)list}$$
$$\xrightarrow{\{\text{get}(\rho),\text{put}(\rho)\}}\ \texttt{(int,}\rho\texttt{)list}$$

[1] For clarity, we here disregard the fact that region inference will also associate a region with the argument pair.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

```
fun split (xs, l, r) : int list * int list =
  case xs of x::y::zs => split(zs,x::l,y::r)
           | [x] => (x::l,r)
           | [] => (l,r)

fun merge (xs,ys) : int list =
  case (xs,ys) of (xs,[]) => xs
                | ([],ys) => ys
                | (l1 as x::xs,l2 as y::ys) =>
                    if x<y then x::merge(xs,l2)
                    else y::merge(l1,ys)

fun msort xs : int list =
  case xs of [] => []
           | [x] => [x]
           | xs => let val (l,r) = split(xs,[],[])
                   in merge(msort l, msort r)
                   end
```

Fig. 1. The Mergesort algorithm on lists.

The region type scheme for `merge` specifies that the function takes as argument a region ρ and a pair of integer lists each of which resides in the region ρ. As a result, the function returns an integer list, which will also reside in the region ρ. The effect $\{\texttt{get}(\rho), \texttt{put}(\rho)\}$, annotated on the function arrow, specifies that, when applied, the function may read from and write into the region ρ. By looking at the body of the `merge` function, we can see that region inference has unified the type of the two argument lists with the type of the result. The reason is that the function contains cases that return each of the arguments. Region inference will result in the following inferred code for the `merge` function:

```
fun merge [ρ] (xs,ys) =
  case (xs,ys) of
    (xs,[]) => xs
  | ([],ys) => ys
  | (l1 as x::xs,l2 as y::ys) =>
     if x<y then op::((x,merge[ρ](xs,l2)) at ρ)
     else op::((y,merge[ρ](l1,ys)) at ρ)
```

The code inferred for the `msort` function also depends on the region type scheme for `split`, which is given as follows:

$$\texttt{val split} : \forall \rho\rho_1\rho_2 .\ (\texttt{int},\rho)\texttt{list} * (\texttt{int},\rho_1)\texttt{list} * (\texttt{int},\rho_2)\texttt{list} \xrightarrow{\{\texttt{get}(\rho),\texttt{put}(\rho_1),\texttt{put}(\rho_2)\}} (\texttt{int},\rho_1)\texttt{list} * (\texttt{int},\rho_2)\texttt{list}$$

Again, for clarity, we have simplified the region type scheme by disregarding the regions for holding the argument triple and the result pair.

Based on the region type schemes for `merge` and `split`, region inference will infer the following code for the `msort` function:

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

```
fun msort [ρ'] (xs: (int,ρ)list) : (int,ρ')list =
  case xs of
    [] => []
  | [x] => op::((x,[]) at ρ')
  | xs => letregion ρ1, ρ2
          in let (l,r) = split [ρ1,ρ2] (xs,[],[])
             in merge [ρ'] (msort [ρ'] l, msort [ρ'] r)
```

The first property to observe is that `msort` may return its result in a region different from its argument list. Second, observe that, for the call to `split`, only two regions are passed as arguments; quantified region variables with only `get`-effects in the function type are not included in the function's formal region parameters. Finally, we observe that, for both recursive calls to `msort`, the region ρ' is passed as region parameter. Whereas polymorphic recursion in regions allow for the recursive `msort` calls to receive its arguments in the local regions ρ_1 and ρ_2, the type of `merge` forces its argument lists to be stored in the same region as its result, which is constrained by the `msort` function to be the region ρ'. As a result, all the intermediate sorted lists pile up in the same region.

A better `msort` implementation will arrange that all intermediate sorted lists are stored in local regions, which can be ensured if we modify the `merge` function slightly:

```
fun copy xs =
  case xs of
    nil => nil
  | x::xs => x::copy xs

fun merge (xs,ys) : int list =
  case (xs,ys) of
    (xs,[]) => copy xs
  | ([],ys) => copy ys
  | (l1 as x::xs,l2 as y::ys) => if x<y then x::merge(xs,l2)
                                 else y::merge(l1,ys)
```

With the added copying, the `merge` function will be inferred to have the following type:

$$\texttt{val merge : } \forall \rho\rho_1\rho_2 .\ \texttt{(int,}\rho_1\texttt{)list * (int,}\rho_2\texttt{)list} \xrightarrow{\{\mathrm{get}(\rho_1),\mathrm{get}(\rho_2),\mathrm{put}(\rho)\}} \texttt{(int,}\rho\texttt{)list}$$

The modified type for the `merge` function will now allow region inference to infer the following code for `msort`:

```
fun msort [ρ'] (xs: (int,ρ)list) : (int,ρ')list =
  case xs of
    [] => []
  | [x] => (op::)((x,[]) at ρ')
  | xs => letregion ρ1, ρ2
          in let (l,r) = split [ρ1,ρ2] (xs,[],[])
             in letregion ρ'1, ρ'2
                in merge [ρ'] (msort [ρ'1] l, msort [ρ'2] r)
```

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Notice that region inference has not unified the four regions ρ_1, ρ_2, ρ'_1, and ρ'_2. Region inference is followed by a series of region analyses, which, in general may benefit from regions not being unified. These analyses include multiplicity inference, which aims at unboxing regions that are inferred to hold at most one value, and storage mode analysis, which aims at emptying a region before it is stored into if it can be inferred that the region contains no live values.

With the *region-friendly* version of `msort`, sorting of a list of length n runs in $O(n)$ space without the use of reference-tracing garbage collection. The execution times (t in seconds), GC times (gc in seconds), and memory usage (m in bytes) of sorting 1,000,000 integers, with the different configurations, are listed in the following table:[2]

	`r`(RI)		`rG`(RI + GENGC)			`rg`(RI + GC)		
	$t(s)$	$m(Mb)$	$t(s)$	$gc(s)$	$m(Mb)$	$t(sec)$	$gc(s)$	$m(Mb)$
`msort-rf`	0.44	103	0.71	0.18	126	0.71	0.15	120
`msort`	0.46	410	1.14	0.44	139	1.31	0.62	139

The most efficient version is the region-friendly version of `msort` compiled and executed without support for reference-tracing garbage collection (row `msort-rf` and columns `r`(RI)). Whereas region inference does well for region-friendly code, the situation is different for the non-region-friendly version of `msort`. Here, the memory usage grows drastically unless reference-tracing garbage collection is added to circumvent the lack of deallocating the intermediate sorted lists. Moreover, we see that the configuration that combines region inference with generational garbage collection (columns `rG`(RI + GENGC)) performs better than when region inference is combined with non-generational garbage collection (columns `rg`(RI + GC)). In particular, the time spent on garbage collection is shorter. It turns out that for the `msort`-case, the `rG` version performs 40 garbage collections of which only 18 of them are major collections, whereas, the `rg` version performs 32 (major) garbage collections.

A thorough evaluation of the different configurations is presented in Section 5.

3 A region type system with typed regions

The integration of region-based memory management with tag-free generational garbage collection that we present in the following sections depends on a number of properties of the underlying evaluation scheme.

In this section, we present a type system that provides us with the necessary guarantees. Compared to the Tofte–Talpin type system (Tofte & Talpin, 1997), the type system that we present ensures that no dangling pointers are introduced during evaluation (Elsman, 2003). Moreover, the type system that we present allow us to give types to regions, which gives us the guarantee that certain values, such as pairs, are allocated in the same regions and that regions containing pairs only contain pairs. This last property is a novel contribution of this work and has not been published elsewhere.

[2] Measurements are averages over 30 runs with a relative standard deviation less than 6.7%. All benchmark programs are executed on a MacBook Pro (15-inch, 2016) with a 2.7GHz Intel Core i7 processor and 16GB of memory running macOS.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

In the remainder of this section, we present a formal treatment for a small ML-like intermediate language extended with region annotations.

3.1 Region types, variables, and effects

A *region type*, ranged over by κ, is either the token `pair`, which classifies regions containing pairs, or the token `other`, which classifies regions containing values other than pairs and integers. Integers are unboxed and thus do not reside in distinguished regions.

We assume a denumerably infinite set of *region variables*, ranged over by ρ. Each region variable has associated with it a region type. We write $rt(\rho)$ to refer to the region type associated with ρ. We also assume a denumerably infinite set of *effect variables*, ranged over by ε, a denumerably infinite set of *type variables*, ranged over by α, and a denumerably infinite set of *program variables*, ranged over by x and f. An *atomic effect*, ranged over by η, is either a region variable or an effect variable. An *arrow effect*, written $\varepsilon.\varphi$, is a pair of an effect variable and a set φ of atomic effects.

Notice that, for simplicity, we do not distinguish between `put`- and `get`-effects in the formal treatment of effects. However, for reasons that we shall make clear later, function types are annotated with arrow effects and not only with effects.

3.2 Types and substitutions

The grammars for *types* (τ), *type and places* (μ), *type schemes* (σ), and *type scheme and places* (π) are as follows:

$$\begin{array}{llll} \mu & ::= (\tau,\rho) \mid \alpha \mid \mathtt{int} & \tau & ::= \mu_1 \times \mu_2 \mid \mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2 \\ \sigma & ::= \forall\vec{\alpha}\vec{\varepsilon}\vec{\rho}.\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2 & \pi & ::= (\sigma,\rho) \mid \mu \end{array}$$

A type scheme and place (or type and place) π is *well-formed* if the sentence $\vdash \pi$ can be derived from the following rules:

$$\vdash \alpha \qquad \vdash \mathtt{int} \qquad \frac{\mathrm{rt}(\rho)=\mathtt{pair}}{\vdash (\mu_1\times\mu_2,\rho)} \qquad \frac{\mathrm{rt}(\rho)=\mathtt{other}}{\vdash (\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2,\rho)} \qquad \frac{\vdash (\tau,\rho)}{\vdash (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau,\rho)}$$

A *region substitution* (S^{r}) is a finite map from region variables to region variables, such that $\mathrm{rt}(\rho)=\mathrm{rt}(S^{\mathrm{r}}(\rho))$ for any region variable $\rho\in\mathrm{dom}(S^{\mathrm{r}})$. A *substitution* ($S$) is a triple $(S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$, where S^{r} is a region substitution, S^{t} is a finite map from type variables to well-formed type and places, and S^{e} is a finite map from effect variables to arrow effects. The effect of applying a substitution on a particular object is to carry out the three substitutions simultaneously on the three kinds of variables in the object (possibly by renaming of bound variables within the object to avoid capture). For effect sets and arrow effects, substitution is defined as follows (Tofte & Birkedal, 2000), assuming $S=(S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$:

$$S(\varphi) = \{S^{\mathrm{r}}(\rho) \mid \rho\in\varphi\} \cup \{\eta \mid \exists\varepsilon,\varepsilon',\varphi'.\varepsilon\in\varphi \wedge S^{\mathrm{e}}(\varepsilon)=\varepsilon'.\varphi' \wedge \eta\in\{\varepsilon'\}\cup\varphi'\}$$

$$S(\varepsilon.\varphi)=\varepsilon'.(\varphi'\cup S(\varphi)),\ \text{where } S^{\mathrm{e}}(\varepsilon)=\varepsilon'.\varphi'$$

One can show that well-formedness is closed under substitution; if $\vdash\pi$ then $\vdash S(\pi)$, for any substitution S.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

We see here why function types are annotated with arrow effects $\varepsilon.\varphi$ and not only with effects φ; with arrow effects, we can make sure that if a non-region-annotated type is given two distinct region annotations, then there exists a substitution, a *unifier*, that, when applied to the two types, will make the two resulting region-annotated types equal. This property is essential for the applied unification-based region inference algorithm (Tofte & Birkedal, 1998), which we shall not discuss further here.

A type scheme $\sigma = \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau'$ *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 $\mathrm{dom}(S^{\mathrm{e}}) = \{\vec{\varepsilon}\}$. 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 $S(\vec{\rho})$.

A *type environment* (Γ) maps program variables to type scheme and places. Following the usual definition of bound variables, we define, for any kind of object o, the *free region variables* and the *free region and effect variables* of o, written $\mathrm{frv}(o)$, and $\mathrm{frev}(o)$, respectively. We write $\mathrm{fv}(o)$ to denote the *free type, region, and effect variables* of o.

3.3 Terms

The grammars for *expressions* (e) and *values* (v) are as follows:

$$
\begin{array}{lcl}
v & ::= & d \mid \langle v_1, v_2\rangle^{\rho} \mid \langle \lambda x.e\rangle^{\rho} \mid \langle \mathtt{fun}\, f\ [\vec{\rho}]\ x = e\rangle^{\rho} \\
e & ::= & v \mid x \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid e_1\ e_2 \mid \lambda x.e\ \mathtt{at}\ \rho \mid \mathtt{letregion}\ \rho\ \mathtt{in}\ e \\
 & \mid & \mathtt{fun}\, f\ [\vec{\rho}]\ x = e\ \mathtt{at}\ \rho \mid e\ [\vec{\rho}]\ \mathtt{at}\ \rho \mid (e_1, e_2)\ \mathtt{at}\ \rho \mid \#i\ e
\end{array}
$$

Values include unboxed integers (d), pairs, ordinary closures, and recursive function closures (which may also take regions as parameters). All values, except integers, are boxed and associated with distinguished regions. An expression can be a value, a variable, a `let`-expression, a function application, a lambda-expression, a `letregion`-construct, a recursive function binding, an application of a recursive function to a list of region parameters, a pair-construct, and a pair-projection expression. Notice that allocating expressions are annotated with an `at`-specifier, which specifies in which region the value should be allocated. The *free (program) variables* of some expression (or value) e is written $\mathrm{fpv}(e)$.

3.4 Typing rules

To guarantee safety of garbage collection, we must ensure that no dangling pointers are introduced during evaluation, which is not guaranteed by the Tofte–Talpin region type system (Tofte & Talpin, 1997). The solution that we apply here is to add additional side conditions to the typing rules for functions that guarantee the absence of dangling pointers (Elsman, 2003).

First, we define a notion of *value containment*; all values in an expression e are contained in a set of regions φ with appropriate region types, if the sentence $\varphi \models_{\mathrm{v}} e$ is derivable from the rules in Figure 2.

We now introduce a relation $\mathcal{G}$, which we shall use to strengthen the typing rules for functions to avoid dangling pointers during evaluation. The relation is derived from the side condition for functions suggested by Tofte and Talpin in (Tofte & Talpin, 1993,

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Values $\boxed{\varphi \models v}$

$$\varphi \models d \qquad \frac{\varphi \models_{\mathrm{v}} e \quad \rho \in \varphi \quad \mathrm{rt}(\rho) = \mathtt{other}}{\varphi \models \langle \lambda x.e \rangle^{\rho}} \qquad \frac{\varphi \models v_1 \quad \varphi \models v_2 \quad \rho \in \varphi \quad \mathrm{rt}(\rho) = \mathtt{pair}}{\varphi \models \langle v_1, v_2 \rangle^{\rho}} \qquad \frac{\rho \in \varphi \quad \varphi \models_{\mathrm{v}} e \quad \mathrm{rt}(\rho) = \mathtt{other}}{\varphi \models \langle \mathtt{fun}\, f\ \mathtt{[}\vec{\rho}\mathtt{]}\ x = e \rangle^{\rho}}$$

Expressions $\boxed{\varphi \models_{\mathrm{v}} e}$

$$\frac{\varphi \models v}{\varphi \models_{\mathrm{v}} v} \qquad \varphi \models_{\mathrm{v}} x \qquad \frac{\varphi \models_{\mathrm{v}} e_1 \quad \varphi \models_{\mathrm{v}} e_2}{\varphi \models_{\mathrm{v}} \mathtt{(}e_1, e_2\mathtt{)}\ \mathtt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \lambda x.e\ \mathtt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \mathtt{\#}i\ e}$$

$$\frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \mathtt{fun}\, f\ \mathtt{[}\vec{\rho}\mathtt{]}\ x = e\ \mathtt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} e\ \mathtt{[}\vec{\rho}\mathtt{]}\ \mathtt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{v}} e_1 \quad \varphi \models_{\mathrm{v}} e_2}{\varphi \models_{\mathrm{v}} e_1\ e_2}$$

$$\frac{\varphi \models_{\mathrm{v}} e_1 \quad \varphi \models_{\mathrm{v}} e_2}{\varphi \models_{\mathrm{v}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2} \qquad \frac{\rho \notin \varphi \quad \varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \mathtt{letregion}\ \rho\ \mathtt{in}\ e}$$

Fig. 2. Value containment.

page 50). The relation $\mathcal{G}$ is parameterised over an environment Γ, a function body e, a set of function parameters X, and the type scheme and place π of the function:

$$\mathcal{G}(\Gamma, e, X, \pi) \;=\; \forall y \in \mathrm{fpv}(e) \setminus X.\mathrm{frev}(\Gamma(y)) \subseteq \mathrm{frev}(\pi) \;\wedge\; \mathrm{frv}(\pi) \models_{\mathrm{v}} e$$

The typing rules for values and expressions are mutually dependent and are shown in Figure 3. The typing rules for values allow inference of sentences of the form $\vdash v : \pi$, which states that "the value v has type scheme and place π". The typing rules for expressions allow inference of sentences of the form $\Gamma \vdash e : \pi, \varphi$, which states that "in the type environment Γ, the expression e has type scheme and place π and effect φ.

The typing rules are closed under substitution; if $\Gamma \vdash e : \pi, \varphi$ then $S(\Gamma) \vdash S(e) : S(\pi), S(\varphi)$, for any substitution S. This property relies on the garbage collection safety relation being closed under substitution.

For simplicity, the typing rule for `let`-bindings does not allow for generalisation.

3.5 A small step dynamic semantics

The dynamic semantics that we present is in the style of a contextual dynamic semantics (Morrisett, 1995) and is similar to the semantics given by Helsen and Thiemann Helsen & Thiemann (2000); Calcagno *et al.* (2002), although it differs in the way that inaccessibility to values in deallocated regions is modeled. Whereas Helsen and Thiemann "null out" references to deallocated regions (to avoid future access), our semantics keep track of a set of currently allocated regions and disallow access to regions that are not in this set.

The grammars for *evaluation contexts* (E) and *instructions* (ι) are shown in Figure 4. Contexts E_φ make explicit the set of regions φ bound by `letregion` constructs that encapsulate the hole in the context.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Values $\boxed{\vdash v : \pi}$

$$\frac{}{\vdash d : \texttt{int}} \qquad \frac{\{x:\mu_1\} \vdash e : \mu_2, \varphi \quad \mu = (\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, \rho) \quad \vdash \mu \quad \mathcal{G}(\{\}, e, \{x\}, \mu)}{\vdash \langle \lambda x.e \rangle^\rho : \mu} \qquad \frac{rt(\rho) = \texttt{pair} \quad \vdash v_1 : \mu_1 \quad \vdash v_2 : \mu_2}{\vdash \langle v_1, v_2 \rangle^\rho : (\mu_1 \times \mu_2, \rho)}$$

$$\frac{\{f : (\forall \vec{\rho}\vec{\varepsilon}.\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \quad \vdash \pi \quad \mathrm{fv}(\vec{\alpha}\vec{\varepsilon}\vec{\rho}) \cap \mathrm{fv}(\varphi) = \emptyset \quad \pi = (\forall \vec{\alpha}\vec{\varepsilon}\vec{\rho}.\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\{\}, e, \{f, x\}, \pi)}{\vdash \langle \texttt{fun}\, f\ [\vec{\rho}]\ x = e \rangle^\rho : \pi}$$

Expressions $\boxed{\Gamma \vdash e : \pi, \varphi}$

$$\frac{\vdash v : \pi}{\Gamma \vdash v : \pi, \emptyset} \qquad \frac{\varphi' \supseteq \varphi \quad \Gamma \vdash e : \pi, \varphi}{\Gamma \vdash e : \pi, \varphi'} \qquad \frac{\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi \quad \mu = (\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, \rho) \quad \vdash \mu \quad \mathcal{G}(\Gamma, e, \{x\}, \mu)}{\Gamma \vdash \lambda x.e\ \texttt{at}\ \rho : \mu, \{\rho\}} \qquad \frac{\Gamma(x) = \pi \quad \vdash \pi}{\Gamma \vdash x : \pi, \emptyset}$$

$$\frac{\Gamma \vdash e : (\sigma, \rho'), \varphi \quad \sigma \geq \tau \text{ via } \vec{\rho} \quad \vdash (\tau, \rho)}{\Gamma \vdash e\ [\vec{\rho}]\ \texttt{at}\ \rho : (\tau, \rho), \varphi \cup \{\rho, \rho'\}} \qquad \frac{\Gamma \vdash e_1 : (\mu' \xrightarrow{\varepsilon.\varphi_0} \mu, \rho), \varphi_1 \quad \Gamma \vdash e_2 : \mu', \varphi_2}{\Gamma \vdash e_1\ e_2 : \mu, \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\varepsilon, \rho\}}$$

$$\frac{rt(\rho) = \texttt{pair} \quad \Gamma \vdash e_1 : \mu_1, \varphi_1 \quad \Gamma \vdash e_2 : \mu_2, \varphi_2}{\Gamma \vdash (e_1, e_2)\ \texttt{at}\ \rho : (\mu_1 \times \mu_2, \rho), \varphi_1 \cup \varphi_2 \cup \{\rho\}} \qquad \frac{i \in \{1, 2\} \quad \Gamma \vdash e : (\mu_1 \times \mu_2, \rho), \varphi}{\Gamma \vdash \#i\ e : \mu_i, \varphi \cup \{\rho\}}$$

$$\frac{\Gamma \vdash e : \mu, \varphi \quad \rho \notin \mathrm{frev}(\Gamma, \mu)}{\Gamma \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e : \mu, \varphi \setminus \{\rho\}} \qquad \frac{\Gamma \vdash e_1 : \pi, \varphi_1 \quad \Gamma + \{x : \pi\} \vdash e_2 : \mu, \varphi_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mu, \varphi_1 \cup \varphi_2}$$

$$\frac{\Gamma + \{f : (\forall \vec{\rho}\vec{\varepsilon}.\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \quad \vdash \pi \quad \mathrm{fv}(\vec{\alpha}\vec{\varepsilon}\vec{\rho}) \cap \mathrm{fv}(\Gamma, \varphi) = \emptyset \quad \pi = (\forall \vec{\alpha}\vec{\varepsilon}\vec{\rho}.\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\Gamma, e, \{f, x\}, \pi)}{\Gamma \vdash \texttt{fun}\, f\ [\vec{\rho}]\ x = e\ \texttt{at}\ \rho : \pi, \{\rho\}}$$

Fig. 3. Typing rules for values and expressions.

The evaluation rules are given in Figure 5 and consist of *allocation and deallocation rules*, *reduction rules*, and a *context rule*. The rules are of the form $e \overset{\varphi}{\hookrightarrow} e'$, which says that, given a set of allocated regions φ, the expression e reduces (in one step) to the expression e'. Next, the *evaluation* relation $\overset{\varphi}{\hookrightarrow}{}^*$ is defined as the least relation formed by the reflexive transitive closure of the relation $\overset{\varphi}{\hookrightarrow}$. We further define $e \Downarrow_\varphi v$ to mean $e \overset{\varphi}{\hookrightarrow}{}^* v$, and $e \Uparrow_\varphi$ to mean that there exists an infinite sequence, $e \overset{\varphi}{\hookrightarrow} e_1 \overset{\varphi}{\hookrightarrow} e_2 \overset{\varphi}{\hookrightarrow} \cdots$.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

$$
\begin{array}{lcll}
E_\varphi & ::= & [\cdot] & (\varphi = \emptyset) \\
& \mid & \texttt{letregion}\ \rho\ \texttt{in}\ E_{\varphi \setminus \{\rho\}} & (\rho \in \varphi) \\
& \mid & E_\varphi\ e \mid v\ E_\varphi \mid E_\varphi\ [\vec{\rho}]\ \texttt{at}\ \rho \mid \texttt{let}\ x = E_\varphi\ \texttt{in}\ e & \\
& \mid & (E_\varphi, e)\ \texttt{at}\ \rho \mid (v, E_\varphi)\ \texttt{at}\ \rho \mid \#i\ E_\varphi & \\
\iota & ::= & d \mid \lambda x.e\ \texttt{at}\ \rho \mid (v_1, v_2)\ \texttt{at}\ \rho & \\
& \mid & \#1\ \langle v_1, v_2\rangle^\rho \mid \#2\ \langle v_1, v_2\rangle^\rho & \\
& \mid & \langle \lambda x.e\rangle^\rho\ v \mid \langle \texttt{fun}\ f\ [\vec{\rho}]\ x = e\rangle^\rho\ [\vec{\rho}\,']\ \texttt{at}\ \rho' &
\end{array}
$$

Fig. 4. The grammars for *evaluation contexts* (E) and *instructions* (ι).

Allocation and Deallocation $\boxed{e \overset{\varphi}{\hookrightarrow} v}$

$$\lambda x.e\ \texttt{at}\ \rho \overset{\varphi \cup \{\rho\}}{\hookrightarrow} \langle \lambda x.e\rangle^\rho \qquad (v_1, v_2)\ \texttt{at}\ \rho \overset{\varphi \cup \{\rho\}}{\hookrightarrow} \langle v_1, v_2\rangle^\rho$$

$$\texttt{fun}\ f\ [\vec{\rho}]\ x = e\ \texttt{at}\ \rho \overset{\varphi \cup \{\rho\}}{\hookrightarrow} \langle \texttt{fun}\ f\ [\vec{\rho}]\ x = e\rangle^\rho \qquad \texttt{letregion}\ \rho\ \texttt{in}\ v \overset{\varphi}{\hookrightarrow} v$$

Reduction and Context $\boxed{e \overset{\varphi}{\hookrightarrow} e'}$

$$\langle \lambda x.e\rangle^\rho\ v \overset{\varphi \cup \{\rho\}}{\hookrightarrow} e[v/x] \qquad \texttt{let}\ x = v\ \texttt{in}\ e \overset{\varphi}{\hookrightarrow} e[v/x]$$

$$\langle \texttt{fun}\ f\ [\vec{\rho}]\ x = e\rangle^\rho\ [\vec{\rho}\,']\ \texttt{at}\ \rho' \overset{\varphi \cup \{\rho\}}{\hookrightarrow} \lambda x.e[\vec{\rho}\,'/\vec{\rho}][(\langle \texttt{fun}\ f\ [\vec{\rho}]\ x = e\rangle^\rho)/f]\ \texttt{at}\ \rho'$$

$$\#1\ \langle v_1, v_2\rangle^\rho \overset{\varphi \cup \{\rho\}}{\hookrightarrow} v_1 \qquad \#2\ \langle v_1, v_2\rangle^\rho \overset{\varphi \cup \{\rho\}}{\hookrightarrow} v_2$$

$$\frac{e \overset{\varphi' \cup \varphi}{\hookrightarrow} e' \quad \varphi \cap \varphi' = \emptyset \quad E_\varphi \neq [\cdot]}{E_\varphi[e] \overset{\varphi'}{\hookrightarrow} E_\varphi[e']}\ [\text{Ctx}]$$

Fig. 5. Evaluation rules.

3.6 Type safety

The proof of type safety is based on well-known techniques for proving type safety for statically typed languages (Morrisett, 1995; Wright & Felleisen, 1994). We shall not present the complete proofs here, but refer the reader to (Elsman, 2003), which includes proofs for a similar system.

We first state a property saying that a well-typed expression is either a value or can be separated into an evaluation context and an instruction:

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

$$\varphi \models_{\mathrm{c}} x \qquad \frac{\varphi \models v}{\varphi \models_{\mathrm{c}} v} \qquad \frac{\rho \notin \varphi \quad \varphi \cup \{\rho\} \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} \texttt{letregion}\ \rho\ \texttt{in}\ e} \qquad \frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} \texttt{let}\ x = e\ \texttt{in}\ e'}$$

$$\frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} e\ e'} \qquad \frac{\varphi \models v \quad \varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} v\ e} \qquad \frac{\varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} e\ [\vec{\rho}]\ \texttt{at}\ \rho}$$

$$\frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} (e, e')\ \texttt{at}\ \rho} \qquad \frac{\varphi \models v \quad \varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} (v, e)\ \texttt{at}\ \rho} \qquad \frac{\varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} \#i\ e}$$

Fig. 6. Context containment.

Proposition 1 (Unique Decomposition). *If $\vdash e : \pi, \varphi$, then either (1) e is a value, or (2) there exists a unique $E_{\varphi'}$, e', and π' such that $e = E_{\varphi'}[e']$ and $\vdash e' : \pi', \varphi \cup \varphi'$ and e' is an instruction.*

Proof. By induction on the structure of e. □

A type preservation property (i.e., subject reduction) for the language, as well as progress and type soundness, can be stated as follows:

Proposition 2 (Preservation). *If $\vdash e : \pi, \varphi$ and $e \xhookrightarrow{\varphi} e'$ then $\vdash e' : \pi, \varphi$.*

Proof. By induction on the derivation $e \xhookrightarrow{\varphi} e'$. □

Proposition 3 (Progress). *If $\vdash e : \pi, \varphi$ then either e is a value or else there exists some e' such that $e \xhookrightarrow{\varphi} e'$.*

Proof. If e is not a value, then by Proposition 1 there exists a unique $E_{\varphi'}$, ι, and π' such that $e = E_{\varphi'}[\iota]$ and $\vdash \iota : \pi', \varphi \cup \varphi'$. The remainder of the proof argues that $\iota \xhookrightarrow{\varphi \cup \varphi'} e''$, for some e'', so that $E_{\varphi'}[\iota] \xhookrightarrow{\varphi} E_{\varphi'}[e'']$ follows from rule [Ctx] in Figure 5. □

Theorem 1 (Type Soundness). *If $\vdash e : \pi, \varphi$, then either $e \Uparrow_{\varphi}$ or else there exists some v such that $e \Downarrow_{\varphi} v$ and $\vdash v : \pi, \varphi$.*

Proof. By induction on the number of rewriting steps, using Proposition 2 and Proposition 3. □

We now introduce the notion of *context containment*, written $\varphi \models_{\mathrm{c}} e$, which expresses that when e can be written in the form $E_{\varphi'}[e']$, values in e' must be contained in the regions in the set $\varphi \cup \varphi'$, where φ' are regions on the stack represented by the evaluation context $E_{\varphi'}$. The definition of context containment is given in Figure 6. The following containment theorem states that, for well-typed programs, containment is preserved under evaluation:

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

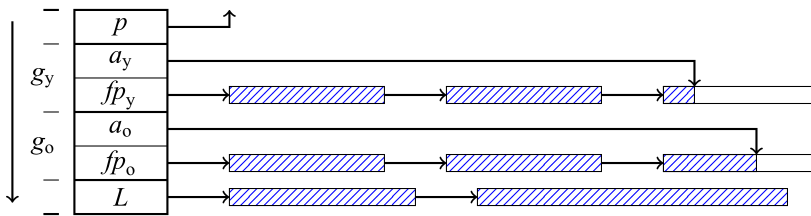

Fig. 7. A region descriptor on the down-growing stack. Region descriptors are linked, through "previous pointers" (p), hold generation descriptors (g_y and g_o), and hold a linked list of large objects (L).

Theorem 2 (Containment). *If* $\vdash e : \pi, \varphi$ *and* $\varphi \models_c e$ *and* $e \overset{\varphi}{\hookrightarrow} e'$ *then* $\varphi \models_c e'$.

Proof. By induction on the structure of e. □

The containment theorem states that evaluation allocates only in regions that are either global or present on the region stack, represented by the evaluation context. Moreover, at any point during evaluation, no region contains a value that does not conform to the region type of the region.

Notice that the specified dynamic semantics does not capture the property that region types are used correctly. That is, the dynamic semantics does not check that pairs, for example, reside in pair regions when a pair element is extracted. For reference-tracing garbage collection, a stronger property is needed, namely that, at any point during evaluation (whenever the garbage collector runs, it must be safe to dereference live reachable values (Morrisett *et al.*, 1995). It is exactly this property that is captured by the containment theorem and that allows a reference-tracing garbage collector to be interleaved with evaluation (as captured by the small-step evaluation semantics).

4 Generational garbage collection

The garbage collector we describe is a *generational* collector, which supports both minor and major collections. In a *minor* collection, only reachable objects allocated in young generations are traversed and *evacuated* (i.e., copied); those allocated in old generations are left untouched. In a *major* collection, all reachable objects are traversed and evacuated. In a minor collection, only reachable objects allocated in young generations are traversed, but a minor collection does not differentiate between in which region an object is stored, as there can be pointers from objects in newer regions to objects in older regions.

A *region descriptor*, which is depicted in Figure 7, represents an unbounded region and consists of a pointer to the previous region descriptor on the stack (p), a generation descriptor for the young generation (g_y), a generation descriptor for the old generation (g_o), and a list (L) for *large objects* (i.e., objects that do not fit in a region page). Each *generation descriptor* (g) consists of a pointer to a list of 1Kb-aligned fixed-sized region pages (fp), each of size 1Kb, and an allocation pointer (a). The treatment of large objects is discussed independently in Section 4.4.

Consider a region r_2 above a region r_1 on the stack, with two generations each. This scenario allows for *deep* pointers from r_2 pointing to objects in region r_1 as shown in Figure 8

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

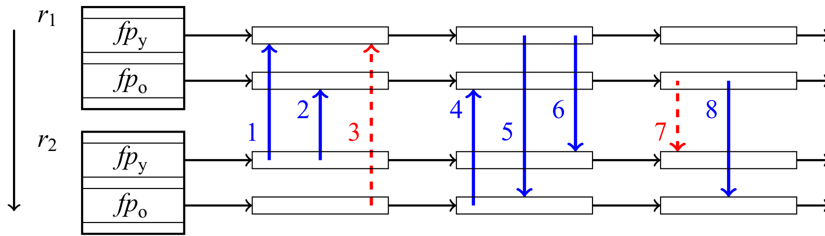

Fig. 8. Possible and impossible pointers. Impossible pointers are those that are dashed. The stack grows downwards.

(labeled 1–4) and *shallow* pointers pointing from objects allocated in region r_1 into objects allocated in region r_2 (labeled 5–8). Shallow pointers only exist between regions allocated in the same `letregion` construct, which is a sufficient requirement to rule out the possibility of dangling pointers (Hallenberg *et al.*, 2002; Elsman, 2003). Shallow pointers (e.g., pointers from values in r_1 to values in r_2) are allowed only between regions that are allocated and deallocated simultaneously. Consider, for instance, a list of type `((int,`ρ_2`)list,`ρ_1`)list`. The outer list's spine is stored in region ρ_1, while the spines of the inner lists are stored in region ρ_2. If region ρ_2 is deallocated before region ρ_1, references from the outer list to the inner lists would become dangling pointers. The scheme that we first describe does not allow for pointers to point from an old generation to a young generation (i.e., the pointers labeled 3 and 7); mutable objects, which may violate this principle, are treated later in Section 4.3. Further notice that newly allocated objects always go in a young generation (pointers 1, 2, 5, and 6). Moreover, pointers 4 and 8 can only be created by the garbage collector.

When an object in a young generation of a region is evacuated, the object may be promoted to the old generation of the region. The collector implements the following promotion strategy, which guarantees that only long-living values are promoted to old generations:

Definition 1 (Promotion Strategy)**.** *Promote objects when they have survived precisely one collection. The first time a value in a region r is evacuated, the value stays allocated in the young generation. During the following garbage collection, the value is promoted (moved) to the old generation of r.*

During a minor garbage collection, objects that have survived one collection must be promoted to the old generation, whereas objects that have not yet survived a collection should remain in the young generation. However, the implementation must preserve a *generation upward-closure* property, which states that, after a collection, whenever a value v has been promoted to an old generation, all values v' pointed to by v are also residing in old generations.

Figure 9 shows two regions and their young generations. The *black areas* contain objects that have survived one collection. The *white areas* signify objects that have been allocated since the last collection. Objects allocated in the black areas will be promoted to an old generation and objects allocated in the white area will stay allocated in a young generation.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

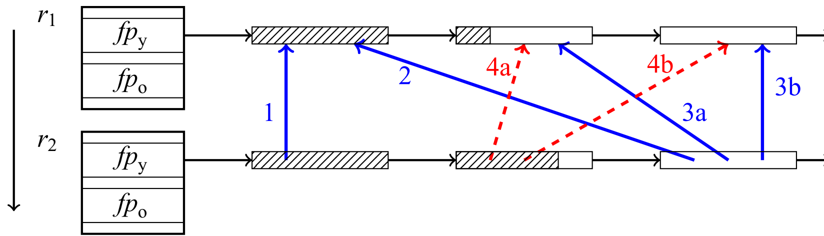

Fig. 9. The black areas contain objects that have survived one collection and white areas contain objects allocated since the last collection.

Figure 9 shows different combinations of pointers from white and black areas into white and black areas.

To implement the promotion strategy, the generation upward-closure invariant must disallow values in black areas to point at values in white areas (pointers 5 and 6 in Figure 9):

Definition 2 (Generation Upward-Closure)**.** *If a value resides in an old generation and points to a value v' then v' resides in an old generation. If a value resides in a black area in a young generation and points to a value v' then v' resides in an old generation or in a black area in a young generation.*

We now argue that the promotion strategy satisfies the Generation Upward-Closure invariant. The argument is a case-by-case analysis of the possible pointers shown in Figure 9 (pointers 1, 2, 3a, and 3b), where each pointer takes the form $v_2 \rightarrow v_1$ and where v_2 is allocated in r_2 and v_1 is allocated in r_1:

Pointer 1. Both v_2 and v_1 reside in black areas, which means that, given v_2 is live, they will both be promoted to old generations according to the promotion strategy. The possibly promoted pointer will thus trivially satisfy Definition 2, part 1.

Pointer 2. If v_2 is live then it will be promoted to the black area of the young generation while v_1 is promoted to the old generation. The possibly promoted pointer will trivially satisfy Definition 2, part 2.

Pointer 3a and 3b. Both v_2 and v_1 reside in white areas of young generations, which means that, given v_2 is live, they will both be promoted to black areas in young generations. Again, the possibly promoted pointer will trivially satisfy Definition 2, part 2.

Pointer 3a gives rise to some considerations because v_1 is allocated in a region page containing both a black and a white area. How do we mark v_1 as being allocated in a white area? One possibility is that we mark each object as being white or black, which will require that all objects are stored with a tag. A less costful solution, which we shall pursue, is to introduce the notion of a *region page color pointer* (*colorPtr*), which points at the first white value in the region page. Given a value v located at a position p in a region page and the color pointer *colorPtr* associated with the region page, if $p < colorPtr$ then v is

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Fig. 10. From-space contains black region pages from old generations, black region pages from young generations, white region pages from young generations, and partly white region pages from young generations. No white region pages from old generations exist.

allocated in the black area of the region page; otherwise, v is allocated in the white area.[3] Notice, that color pointers are updated and referenced only during a garbage collection; it does not change when allocating new values.

For the scheme to be sound, we need to make sure that pointers of the form of pointer 4a and pointer 4b never occur as the promotion strategy would otherwise lead to pointers from old generations to young generations, which would violate Definition 2. As we have shown, the garbage collector will never introduce such pointers and, neither will the mutator, except due to mutable data assignment, which we will treat in Section 4.3.

An alternative to the implemented promotion strategy is to add additional generations and let a minor collection traverse all objects except those in an oldest generation. Such a solution, however, could introduce a large amount of unused memory in region pages. Another promotion strategy would be to promote objects when they have survived a number ($N \geq 0$) of collections, which generalises the implemented promotion strategy, but is intractable as it requires tracking of the number of times each object in a young generation has survived a collection.

4.1 Evacuating objects

The *evacuation* process copies live objects into fresh pages so that the copied-from pages can be reclaimed, including the parts of the pages that hold unreachable values. Definition 2 is implemented as follows. During a major collection, the collector will evacuate objects from old generations into old generations. During a minor collection, however, old generations will be left untouched and the collector will not attempt to traverse values stored in old- generation pages. During a major or a minor collection, the collector will evacuate objects in young-generation white areas into young-generation black areas. Moreover, the collector will evacuate objects in young-generation black areas into old generations. The evacuation strategy is implemented by marking all region pages in old generations black, which means that the same algorithm can be used to evacuate objects in minor and major collections. All objects in black areas are copied into black areas in old generations. All objects in white areas are copied into black areas in young generations. All objects allocated between two collections are allocated in white areas in young generations.

Before a major collection, all region pages are assembled to form the from-space as shown in Figure 10. For a minor collection, from-space is formed from all young-generation pages. After a collection (minor or major), the from-space pages are added to the free list of pages.

To distinguish pointers from non-pointers, integers and other unboxed values (e.g., booleans and enumeration datatypes) are represented as tagged values with the least significant bit set. Records are represented as a vector of values with a prefix tag word,

[3] In the implementation, the color pointer associated with a region page is located in the header of the page. If *colorPtr* points past the page, the entire page is black.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

which is used by the collector to identify the number of record components. Pairs and triples, however, are represented without a prefix tag word. Given a pointer to a value in a region page, the collector can determine that the value is a pair or a triple by inspecting the region type associated with the region in which the object resides. In practice, the implementation works with the region types RTY_BOT, RTY_PAIR, RTY_TRIPLE, RTY_DOUBLE, RTY_REF, RTY_ARRAY, and RTY_TOP. Here, the region type RTY_TOP is used for specifying regions that can contain values of arbitrary type, except those associated with the other region types. The region type RTY_BOT never occurs at run time, but is used for specifying type and region-polymorphic functions. The region unification algorithm will fail to unify two regions with different types (except if one of the region types is RTY_BOT), which provides the guarantee that values stored in a region at run time are classified according to the region type of the region. For efficiency, the region type for a region is stored both in the generation descriptor for the old generation and in the generation descriptor for the young generation.

We mentioned earlier that a region representation analysis aims at identifying so-called *finite regions*, which are regions that have been inferred to hold at most one value (Birkedal *et al.*, 1996). In the implementation, values stored in finite regions on the stack are traversed by the garbage collector, but never copied or collected.

4.2 The GC algorithm

The GC algorithm makes use of a series of auxiliary utility functions:

- `in_oldgen_and_minor(p)`: Returns TRUE iff the collection is a minor collection and `p` points to an object in a region page for which the old-generation bit is set.
- `is_int(p)`: Returns TRUE iff the least-significant bit in `p` is set.
- `tag_is_fwd_ptr(w)`: Returns TRUE iff the tag word `w` is the reserved forward pointer tag, which is different from other tags used for tagged objects.
- `is_pairregion(r)`: Returns TRUE iff the runtime type associated with the region descriptor `r` is REGION_PAIR.
- `in_tospace(p)`: Returns TRUE iff `p` points to an object in a region page for which the to-space bit is set.
- `acopy_pair(r,p)`: Allocates a pair in the region associated with the region descriptor `r` and copies into the newly allocated memory the two pointers (or integers) contained in the pair pointed to by `p`.
- `obj_sz(w)`: Returns the size of the object in words, given its tag word.
- `gendesc(p)`: Returns the generation descriptor for the generation in which the object pointed to by `p` resides. Each region page in the generation has associated with it a generation pointer, pointing at the generation descriptor for the generation. Generation pointers are installed when a new region page is associated with a generation.
- `push_scanstack(a)`: Pushes the allocation pointer `a` onto the scan stack.
- `pop_scanstack()`: Pops and returns the top scan pointer from the scan stack. Returns NULL if the scan stack is empty.
- `target_gen(g,p)`: Returns the old generation associated with g's region unless g is a young generation and p appears in a white area in g, in which case it returns g.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

```
void* evacuate(void* p) {
  if ( is_int(p) || in_oldgen_and_minor(p) ) { return p; }
  g = gendesc(p);
  gt = target_gen(g,p);
  if ( is_pairregion(g) ) {
    if ( in_tospace(*(p+1)) ) { return *(p+1); } // fwd-ptr
    a = acopy_pair(gt,p);
    *(p+1) = a;                                   // set fwd-ptr
  } else {
    if ( tag_is_fwd_ptr(*p) ) { return *p; }
    a = acopy(gt,p);
    *p = a;                                       // set fwd-ptr
  }
  if ( gt->status == NONE ) {
    gt->status = SOME;
    push_scanstack(a);
  }
  return a;
}
```

Fig. 11. The function `evacuate` assumes that the argument `p` points to an object and that it perhaps resides in from-space and needs to be copied to to-space. After copying, a forward-pointer is installed.

A central part of the GC algorithm is the function `evacuate`, shown in Figure 11, which copies live values under consideration from from-space into to-space. It takes a pointer `p` and copies the value pointed to into to-space provided it is not already copied and that it is a prospect (i.e., under a minor collection, values in old generations are not copied.) For brevity, only pairs are treated specially; the implementation also treats regions of type RTY_TRIPLE and RTY_REF specially, as also triples and references are represented untagged.

Another central function is the `cheney` function, which takes care of scanning the values that have been copied into to-space. During scanning, the `cheney` function may call `evacuate` on values that have themselves not yet been copied, which may cause an update to the generation allocation pointer. Once, for all regions, the scan pointer reaches the allocation pointer, the collection terminates. The `cheney` function is shown in Figure 12. Notice, again, that special treatment is required for dealing with untagged values (only the case for pairs is shown.)

The main GC function, called `gc` is shown in Figure 13. It evacuates all values in the root set and continues by calling the `cheney` function on all values on the scan stack. Notice that the `evacuate` function pushes values that have been copied to to-space onto the scan stack for further processing (the g_t`->status` field is used to ensure that the scan pointer is pushed at most once.)

To determines whether a minor or a major collection is run, a so-called *heap-to-live ratio* is maintained, which by default is set to 3.0. Whenever the length of the free list of pages becomes less than one-third of the total number of region pages that make up the heap, garbage collection is initiated upon the next function entry (i.e., safe point). After

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

```
void cheney(void* s) {
  g = gendesc(s);
  if ( is_pairregion(g) ) {
    for ( ; s+1 != g->a ; s = next_pair(s,g) ) {
      *(s+1) = evacuate(*(s+1));
      *(s+2) = evacuate(*(s+2));
    }
  } else {
    for ( ; s != g->a ; s = next_value(s,g) ) {
      for ( i=1; i<obj_sz(*s); i++ ) {
        *(s+i) = evacuate(*(s+i));
      }
    }
  }
  g->status = NONE;
}
```

Fig. 12. The function `cheney` assumes that the argument scan pointer `s` points to a value that has already been copied to to-space but for which the components have not yet been evacuated. The function is named `cheney` because it degenerates to Cheney's algorithm if multi-generations are disabled.

```
void gc(void** rootset) {
  while ( p = next_root(rootset) ) { *p = evacuate(*p); }
  while ( p = pop_scanstack() ) { cheney(p); }
}
```

Fig. 13. The main GC function evacuates each of the values in the root set after which the `cheney` function is called with scan pointers from the scan stack as long as there are scan pointers on the stack.

each collection, it is ensured that the number of allocated region pages is at least 3.0 times the number of region pages that make up to-space (given the heap-to-live ratio is 3.0). The following rules are deployed for switching between major and minor collections, allowing an arbitrary number of minor collections between two major collections:

1. If the current collection is a major collection, the next collection will be a minor collection. The region heap is enlarged to satisfy the heap-to-live ratio.
2. If the current collection is a minor collection and the heap-to-live ratio is not satisfied after the collection, the next collection will be a major collection.

4.3 Mutable objects

In the presence of mutable objects, the generation upward closure invariant may be violated during program evaluation. In particular, a reference cell (which are rare in a functional language) residing in an old generation, may be assigned to point at a value residing in a young generation. We refine the generation upward-closure condition as follows:

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Definition 3 (Refined Generation Upward-Closure). *For all values v, if v is non-mutable and resides in an old generation then for all values v' pointed to from v, v' resides in an old generation.*

The refined generation upward-closure invariant is safe, if each minor collection traverses all reachable mutable values (even those that reside in old generations). For minor collections we extend the root set to contain, not only live values on the stack, but also all references and arrays allocated. How does the collector locate all references and arrays? Simply by arranging that such values are stored in regions with distinguished region types. During a minor collection, the region stack is traversed and objects in regions of type RTY_REF and RTY_ARRAY are traversed. Thus, we avoid the cost of a write barrier at each reference assignment and the implementation of the usual "remembered set" of mutable values that have been updated since the previous collection. This strategy can potentially be more costly than if a proper "remembered set" is maintained, which we leave to future work. A possible compromise could be to allow for a bit in a region page descriptor (or in a region descriptor) for regions of type RTY_REF and RTY_ARRAY to record whether a value in the page (or in the region) has been modified since the previous collection. For applications that make extensive use of mutable data structures, such as certain implementations of unification and graph algorithms (a good example is the unification-based region inference algorithm applied in the MLKit), this strategy introduces a small mutator overhead (a write-barrier operation at each reference assignment), but may decrease the time used in minor collections.

4.4 Large objects

Concerning the treatment of large objects, there are several options. In the implementation, we are currently treating large objects without dividing them into young and old objects. Large objects are kept in one list associated with a region descriptor. Following this strategy, large objects are not associated with a particular generation (nor need they be associated with a color) and may therefore be collected only during major collections. However, following this scheme, large objects are traversed (not copied), when reached, both during major and minor collections. A special account of the number of bytes allocated in large objects, before and after major collections, makes it possible for garbage collection to be triggered due to allocations of large objects (and the setting of the heap-to-live ratio).

Two alternatives to dealing with large objects would be to have two lists of large objects, one for each of the two generations, or one list where each object is annotated with generation information.

5 Experimental results

In this section, we describe a series of experiments that serve to demonstrate the relationship between region inference, non-generational garbage collection, and the generational garbage collection algorithm presented in Section 4.

The experiments are performed with MLKit version 4.5.1 and MLton v20201023, which both generate native x64 machine code. The two compilers are very different, however.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Whereas MLton is a whole-program highly optimising compiler, MLKit features a smart-recompilation system that allows for quick rebuilds upon modification of source code (Elsman, 2008).

All benchmark programs are executed on a MacBook Pro (15-inch, 2016) with a 2.7GHz Intel Core i7 processor and 16GB of memory running macOS. Times and memory usage reported are measured using `getrusage` (BSD System Call) and the macOS `/usr/bin/time` program. Measurements are averages over 30 runs. We use m to specify memory usage (resident set size) and t to specify execution time (in seconds). Subscripts describe the mode of the compiler, with $*_r$ signifying region inference enabled, $*_g$ signifying garbage collection enabled, and $*_G$ signifying generational garbage collection enabled. Thus, t_{rG} specifies execution time with region inference and generational garbage collection enabled. We use m_{mlton} and t_{mlton} to signify memory usage and execution time for executables running code generated by MLton. The benchmark programs span from micro-benchmarks such as `fib37` and `tak` (7 and 12 lines), which only use the runtime stack for allocation, to larger programs, such as `vliw` and `mlyacc` (3,676 and 7,353 lines), that solve real-world problems. The program `msort-rf` is a region-friendly version of Mergesort for which the merge function allocates its result in a region different from its arguments (by copying the list tails). With no sharing between its arguments and its result, the use of the merge function in the Mergesort algorithm leads to good region memory performance due to region-polymorphic recursion, as shown in Section 2.

By *disabling* region inference, we mean instructing region inference to allocate all values that would be allocated in infinite regions in global regions (collapsed according to their region type). Then not a single infinite region is deallocated at run time and the non-generational garbage collection algorithm essentially reduces to Cheney's algorithm. Disabling region inference does not change the property that many values are allocated in finite regions on the stack.

5.1 Comparison of execution times

In this section, we compare the execution time for the different benchmark programs compiled with MLton and with different configurations of the MLKit compiler. Figure 14 shows execution time for the benchmark programs compiled with the different compiler configurations.

We see that for most of the programs, MLton outperforms the MLKit (with and without garbage collection enabled). MLton's whole-program compilation strategy, efficient IO operations, and optimised instruction selection for the x64 architecture, are good candidates for an explanation. For a few of the examples (i.e., `DLX`, `fib37`, `msort`, `msort-rf`, and `simple`), MLton is outperformed by MLKit's region-inference only configuration (`r`), and, in a few cases, also when region inference is combined with garbage collection.

Here are some of the conclusions that we can draw from comparing the MLKit configurations:

1. GC adds an execution-time overhead. In all cases (except for the `life` benchmark), combining region inference with a garbage collection mechanism has an execution-time overhead compared to when region inference is used alone. The region-inference only configuration, however, sometimes leads to the use of an

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

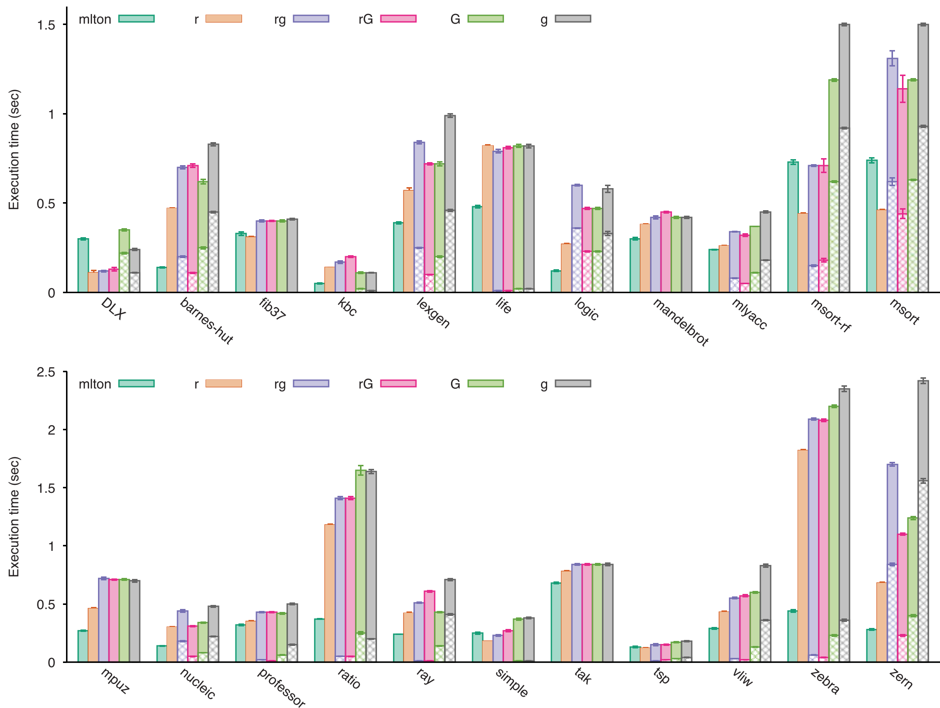

Fig. 14. Execution times for MLKit-generated executables compared to execution times for MLton generated executables. For each vertical bar, markers specify the absolute standard deviation for the corresponding 30 runs. GC times are visualised as shaded areas, which are also marked with the absolute standard deviation.

excessive amount of memory (as we shall see in the next section), which makes the configuration unendurable in practice for non-region-optimised programs.

2. The GC-only configurations (i.e., `g` and `G`) most often involve an overhead compared to the other MLKit configuration. There are a few cases where a GC-only configuration performs slightly better than when a garbage collection strategy is combined with region inference (i.e., `barnes-hut`, `kbc`, and `ray`). For these cases, region inference introduces an overhead by leading to the management of an excessive number of regions at runtime (e.g., functions are passed a large number of regions as parameters).
3. Configuration `rG`, which combines region inference and generational garbage-collection, performs as good or better than configuration `rg`, which combines region inference with ordinary garbage collection, for almost all benchmarks. Exceptions include the benchmarks `kbc`, `mandelbrot`, `ray`, and `simple`, for which we see a small overhead caused by the programs having to manage multiple generations for each region.
4. For a number of the benchmarks, configuration `rG` performs significantly better than configuration `rg` (i.e., `lexgen`, `logic`, `mlyacc`, `msort`, `nucleic`, and `zern`), which is primarily caused by shorter accumulated garbage collection times; we shall discuss the aspect of accumulated garbage collection times further in Section 5.3.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Program	MLton		r(RI)		rg(RI+GC)		rG(RI+GENGC)		G(GENGC)		g(GC)	
	t(*s*)	*m*(*Mb*)	*t*(*s*)	*m*(*Mb*)	*t*(*s*)	*m*(*Mb*)	*t*(*s*)	*m*(*Mb*)	*t*(*s*)	*m*(*Mb*)	*t*(*s*)	*m*(*Mb*)
`DLX`	0.30	31.7	†0.11	6.1	0.12	6.3	†0.13	6.5	0.35	8.1	0.24	9.2
`barnes-hut`	0.14	1.9	0.47	**185.0**	0.70	2.3	0.71	2.5	0.62	2.2	0.83	2.2
`fib37`	0.33	1.0	0.31	1.1	0.40	1.1	0.40	1.1	0.40	1.1	0.41	1.1
`kbc`	0.05	2.1	0.14	6.8	0.17	3.3	0.20	4.1	0.11	2.1	0.11	2.1
`lexgen`	0.39	18.4	0.57	**73.0**	0.84	6.9	0.72	†8.5	0.72	†6.0	0.99	†5.9
`life`	0.48	2.0	0.82	**14.4**	0.79	1.5	0.81	1.6	0.82	1.4	0.82	1.4
`logic`	0.12	2.1	0.27	**277**	0.60	2.5	0.47	2.5	0.47	2.5	0.58	2.5
`mandelbrot`	0.30	0.8	0.38	1.3	0.42	1.4	0.45	1.4	0.42	1.4	0.42	1.4
`mlyacc`	0.24	10.8	0.26	**127.0**	0.34	6.9	0.32	7.8	0.37	6.9	0.45	6.0
`mpuz`	0.27	0.9	0.46	1.1	0.72	1.1	0.71	1.2	0.71	1.1	0.70	1.1
`msort-rf`	0.73	583.0	0.44	103.0	0.71	120.0	†0.71	126.0	1.19	139.0	1.50	138.0
`msort`	0.74	424.0	0.46	**410.0**	1.31	139.0	†1.14	139.0	1.19	139.0	1.50	144.0
`nucleic`	0.14	2.0	0.30	**266.0**	0.44	2.6	0.31	2.7	0.34	2.4	0.48	2.4
`professor`	0.32	1.3	0.35	**10.2**	0.43	1.2	0.43	1.3	0.42	1.1	0.50	1.1
`ratio`	0.37	47.3	1.18	38.4	1.41	10.6	1.41	10.6	1.65	10.6	1.64	10.5
`ray`	0.24	14.0	0.42	12.7	0.51	6.3	0.61	6.7	0.43	6.0	0.71	5.9
`simple`	0.25	7.4	0.18	2.3	0.23	2.7	0.27	2.8	0.37	3.3	0.38	3.3
`tak`	0.68	0.8	0.78	1.0	0.84	1.0	0.84	1.0	0.84	1.0	0.84	1.0
`tsp`	0.13	10.5	0.12	6.2	†0.15	10.4	0.15	9.1	0.17	13.3	0.18	14.2
`vliw`	0.29	9.0	0.43	**45.4**	0.55	5.2	0.57	7.0	0.60	4.8	0.83	4.6
`zebra`	0.44	1.3	1.82	**133.0**	2.09	1.2	2.08	1.4	2.20	1.1	2.35	1.1
`zern`	0.28	10.5	0.68	4.3	1.70	3.7	1.10	3.8	1.24	4.8	2.42	3.7

Fig. 15. Execution times (in seconds) and memory usage (in Mb) for executables generated with MLton and different configurations of MLKit.

5.2 Comparison of memory usage

In this section, we present the memory usage for the different configurations and discuss the memory usage in relation to the execution time for the various benchmarks and configurations. Raw numbers for the configurations are shown in Figure 15. Again, measurements reported are averages over 30 runs. The relative standard deviation varies between 0% and 11.9%; only eight measurements (those measurements annotated with a dagger (†) in Figure 15) have a relative standard deviation exceeding 4.3%.

When comparing the MLKit configurations, we see that even though the time performance of all benchmarks are better with region inference alone (i.e., the `r` configuration), for some of the benchmark programs (i.e., those with numbers marked in bold in Figure 15), region inference alone does not suffice to obtain good memory performance. For a few of the benchmarks, the `r` configuration results in better memory usage than any of the other configurations. Possible reasons for this behavior are (1) that the `r` configuration introduces less fragmentation than the other MLKit configurations (each region contains only one list of region pages) and (2) that the `r` configuration permits dangling pointers.

Another important observation that we shall mention here is that the `rG` configuration does not result in a drastically larger memory usage than the `rg` configuration. In Section 5.4, we shall investigate the fragmentation issues caused by region inference dividing memory into pages and generations dividing regions into multiple page lists.

For the MLKit configurations that make use of garbage collection, a heap-to-live ratio of 3.0 is used, which is the default. The default heap-to-live ratio used by MLton is 8.0, which explains the larger memory usage for many of the MLton-compiled executables, compared to MLKit-compiled executables. Moreover, as we shall see in Section 5.5, it turns out that

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Program	c_{rg}	g_{rg} (s)	p_{rg} (%)	c_{rG}		g_{rG}(s)		p_{rG} (%)	c_g	g_g(s)	c_G		g_G(s)	
DLX	5	0.00	0	8	(4)	0.00	(0.00)	0	104	0.11	205	(102)	0.22	(0.10)
barnes-hut	1645	0.20	63	1393	(440)	0.11	(0.06)	63	3861	0.45	3818	(1210)	0.25	(0.15)
fib37	1	0.00	0	2	(1)	0.00	(0.00)	5	1	0.00	2	(1)	0.00	(0.00)
kbc	37	0.00	44	33	(12)	0.00	(0.00)	43	225	0.01	305	(64)	0.02	(0.00)
lexgen	485	0.25	81	343	(30)	0.10	(0.01)	79	866	0.46	888	(109)	0.20	(0.05)
life	147	0.01	17	132	(19)	0.01	(0.00)	17	789	0.02	938	(167)	0.02	(0.01)
logic	2610	0.36	100	2696	(432)	0.23	(0.06)	100	2574	0.33	2680	(434)	0.23	(0.06)
mandelbrot	1	0.00	0	2	(1)	0.00	(0.00)	3	1	0.00	2	(1)	0.00	(0.00)
mlyacc	160	0.08	63	178	(29)	0.05	(0.01)	63	421	0.18	445	(115)	0.11	(0.06)
mpuz	2	0.00	2	4	(2)	0.00	(0.00)	2	2	0.00	3	(1)	0.00	(0.00)
msort-rf	21	0.15	4	30	(15)	0.18	(0.12)	7	40	0.92	49	(22)	0.62	(0.41)
msort	32	0.62	57	40	(18)	0.44	(0.26)	53	40	0.93	49	(22)	0.63	(0.41)
nucleic	1751	0.18	94	1506	(102)	0.05	(0.01)	94	2253	0.22	2412	(233)	0.08	(0.03)
professor	1772	0.02	28	1086	(33)	0.01	(0.00)	27	16675	0.15	14025	(766)	0.06	(0.01)
ratio	34	0.05	24	40	(10)	0.05	(0.01)	23	106	0.20	151	(30)	0.25	(0.05)
ray	22	0.01	3	28	(12)	0.01	(0.01)	3	523	0.41	425	(73)	0.14	(0.06)
simple	7	0.00	3	10	(5)	0.00	(0.00)	5	18	0.01	23	(10)	0.01	(0.00)
tak	1	0.00	0	1	(0)	0.00	(0.00)	0	1	0.00	1	(0)	0.00	(0.00)
tsp	11	0.01	2	18	(9)	0.02	(0.01)	2	17	0.04	25	(12)	0.03	(0.01)
vliw	93	0.03	13	54	(13)	0.02	(0.00)	12	1088	0.36	1136	(135)	0.13	(0.05)
zebra	5009	0.06	63	3008	(55)	0.04	(0.00)	63	39044	0.36	31033	(6007)	0.23	(0.06)
zern	84251	0.84	97	53436	(4)	0.23	(0.00)	91	158546	1.56	112904	(102)	0.40	(0.00)

Fig. 16. GC counts (c_*) and GC times (g_*) for the different configurations. Reported counts are the total number of collections with the number of major collections and the accumulated major collection time in parentheses. The p_* columns show the percentage of bytes reclaimed by GC (in contrast to region inference).

we can improve the performance of benchmarks with high garbage collection times by adjusting the heap-to-live ratio

5.3 Generational garbage collection

By looking at the performance graphs in Figure 14 and the performance numbers in Figure 15, we see that generational garbage collection alone (without region inference) performs better than or equivalent to non-generational garbage collection, except in one case (i.e., benchmark DLX). We shall return to this case when we report on the number of major collections versus the number of minor collections performed for the benchmarks in each configuration.

Figure 16 shows the garbage collection counts (c_{rg}, c_{rG}, c_g, and c_G) for the different configurations. Notice that the garbage collection counts (and times) are smaller when region inference is enabled, which indicates that region inference takes care of a large amount of memory recycling, which again means that the reference-tracing garbage collection is triggered less often. Notice also that, for the configurations that combine region inference and garbage collection, the accumulated garbage collection time for the configuration using generational collection (column g_{rG}) is smaller than or identical to the accumulated garbage collection time for the configuration using non-generational garbage collection (column g_{rg}), for all benchmarks, except for the benchmark `msort-rf` for which generational garbage collection introduces a small overhead (less than 20% in accumulated garbage collection time). However, the improvements are often significant, which can also be seen in Figure 14. For the benchmarks `barnes-hut`, `lexgen`, `nucleic`, and `zern`, the

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

improvement in accumulated garbage collection time is about 50% or larger, which are caused by a significant decrease in the number of major collections. For other benchmarks, including `logic`, `mlyacc`, `msort`, and `zebra`, we see a more modest improvement in accumulated garbage collection time (between 20% and 50%), again caused by a decrease in the number of major collections.

Finally, notice that the percentage of memory reclaimed by the garbage collector is (close to) invariant to whether the garbage collector is generational or not (p_{rg} versus p_{rG}).

The MLKit features a *region profiling tool* (Hallenberg, 1996), which allows for showing a program's use of regions over time. Figure 17 shows region profiles of MLYacc computations for four different MLKit runtime configurations. The profiles show that generational garbage collection combined with region inference often requires more memory than when region inference is combined with non-generational garbage collection, but also, that the profile obtained alone with generational garbage collection is similar to the profile obtained with region inference and non-generational garbage collection enabled. The figure also demonstrates a crucial point, namely that the global regions are often those that need to be collected by the reference tracing collector, which means that schemes that attempt to collect only the topmost regions will probably fail to be effective.

5.4 Memory waste

Figure 18 reports memory waste percentages (percentages of unused memory in region pages) for the configurations w_{rg} (region inference and non-generational garbage collection), w_{rG} (region inference and generational garbage collection), w_{g} (non-generational garbage collection), and w_{G} (generational garbage collection).[4] Notice the (at first) mysterious numbers for the benchmarks `fib37` and `mandelbrot`, which, for all configurations allocate very few objects in infinite regions, which are therefore almost empty.

As expected, the waste is high for the region inference configurations. Moreover, region inference combined with generational garbage collection results in more memory waste (unused memory in region pages) than when combined with non-generational garbage collection (up to 17% points more, not considering the `fib37` and `mandelbrot` benchmarks). The reason is that, with generational garbage collection, each infinite region contains two lists of region pages (one list for each generation), each of which may not be fully utilised. As mentioned earlier, each region page is fixed at size 1Kb. A different page size would most likely lead to different percentage waste numbers; smaller pages would lead to internal fragmentation, while larger pages would have more last-page underutilisation.

5.5 Adjusting the heap-to-live ratio

For a few of the benchmarks, we see that a significant part of the execution time is spent during garbage collection in the configurations that make use of garbage collection. Particularly, for the `rG` configuration, we see from Figure 14 and Figure 16 that the accumulated garbage collection times for the `logic` and `msort` programs account for 49%

[4] The memory waste that we account for here only relates to unused memory in region pages. A different kind of memory waste, which we do not account for, has to do with dead values in old generations. Such waste is eliminated, however, whenever a major garbage collection is run.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

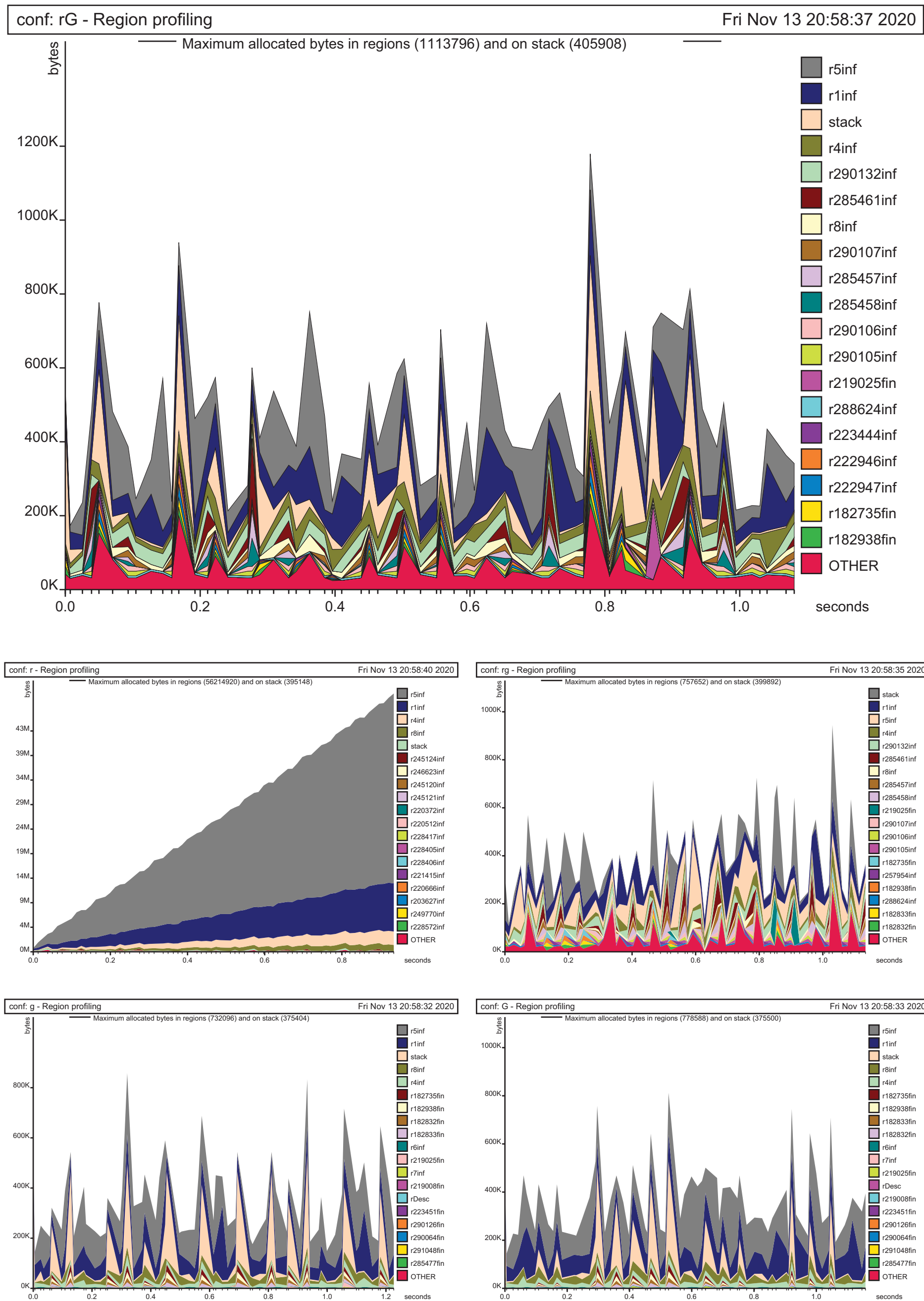

Fig. 17. Region profiles showing memory usage over time for different runtime configurations (top: `rG`, middle-left: `r`, middle-right: `rg`, bottom-left: `g`, and bottom-right: `G`). Notice that, for the `g` and `G` configurations, there are no non-global infinite regions. Notice also that, without reference-tracing garbage collection (configuration `r`), the values in global regions are never freed.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Program	w_{rg} (%)	w_{rG} (%)	w_{g} (%)	w_{G} (%)
DLX	26	42	1	2
barnes-hut	12	20	2	5
fib37	0	91	0	91
kbc	41	57	5	10
lexgen	8	16	1	2
life	8	18	5	9
logic	3	6	3	6
mandelbrot	0	91	0	91
mlyacc	7	18	1	2
msort-rf	5	11	3	7
msort	4	8	3	7

Program	w_{rg} (%)	w_{rG} (%)	w_{g} (%)	w_{G} (%)
mpuz	69	86	46	79
nucleic	12	19	3	5
professor	31	39	16	25
ratio	6	9	1	2
ray	9	17	1	2
simple	17	31	4	9
tak	0	0	0	0
tsp	10	16	7	13
vliw	11	19	1	2
zebra	30	38	22	30
zern	1	2	0	0

Fig. 18. Memory waste. The numbers show the average percentage of region waste (unused memory in region pages) measured at each collection.

and 39%, respectively. The following table reports the total execution time and the accumulated garbage collection time for the two benchmark programs with a heap-to-live ratio of 3 (as reported earlier) and a heap-to-live ratio of 8:

	Heap-to-live ratio $=3$			Heap-to-live ratio $=8$		
Program	$t_{rG}(s)$	$g_{rG}(s)$	$m_{rG}(Mb)$	$t_{rG}(s)$	$g_{rG}(s)$	$m_{rG}(Mb)$
logic	0.47	0.23	2.5	0.28	0.05	2.9
msort	1.14	0.44	140	0.77	0.28	201

We see that, in these two cases, a larger heap-to-live ratio decreases the overall execution time, while the overall memory usage is increased.[5]

6 Related work

This paper is an extended version of the paper "On the Effects of Integrating Region-Based Memory Management and Generational Garbage Collection in ML", which appeared in the proceedings of the PADL '20 symposium (Elsman & Hallenberg, 2020). The present paper includes a presentation of the theoretical foundations of the work and an extended empirical evaluation. In particular, the paper includes a novel theoretical justification that, using typed regions, region- annotated intermediate expressions can be guaranteed to have the property that regions will only contain values of the same type. This property is essential for allowing a tag-free representation of pairs, triples, and references, which provides dramatic savings on allocated memory and execution time.

Most related to this work is the previous work on combining region inference and garbage collection in the MLKit (Hallenberg *et al.*, 2002). Compared to the earlier work,

[5] Notice that the overall memory usage, measured in maximum resident set size (determined by the underlying operating system) is not in a simple way related to the heap-to-live ratio, which relates directly to the number of allocated region pages.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

the present work investigates how generational garbage collection can be combined with region inference and how the concept of typed regions can be used to circumvent the need for a generation write barrier. In particular, without using a generation write barrier, typed regions allow for the garbage collector to apply a conservative strategy that scans only a small fraction of the old-generation heap during a minor collection (i.e., regions containing mutable data). There is a large body of related work concerning general garbage collection techniques (Jones *et al.*, 2011) and garbage collection techniques for functional languages, including (Doligez & Leroy, 1993; Reppy, 1994; Huelsbergen & Winterbottom, 1998; Ueno & Ohori, 2016).

Incremental, concurrent, and real-time garbage collection techniques for functional languages have recently obtained much attention. In particular, the presence of generations has been shown useful for collecting parts of the heap incrementally and in a concurrent and parallel fashion (Marlow *et al.*, 2009; Anderson, 2010; Marlow & Peyton Jones, 2011). We leave it to future work to investigate the use of regions and generations in the MLKit for supporting concurrency and parallelism in the language.

There are a series of proposals for tag-free garbage collection schemes (Appel, 1989; Goldberg, 1991; Goldberg & Gloger, 1992; Aditya *et al.*, 1994; Tolmach, 1994) and nearly tag-free garbage collection schemes (Morrisett *et al.*, 1996; Tarditi *et al.*, 1996).

The *partly tag-free garbage collection* scheme that we present here does not involve untagging of all values. In particular, unboxed objects (e.g., integers and booleans) are tagged in our system, which makes it possible to distinguish pointers from unboxed objects at runtime. However, the scheme allows for commonly used data structures, such as tuples, reals, and reference cells, to be untagged, which, as mentioned, can lead to significant time and memory savings, in particular because pairs and triples are used for the implementation of many dynamic data structures, including lists and trees.[6]

As is the case for other techniques that support full untagging, our technique does not involve traversing the runtime stack to determine types during garbage collection (Appel, 1989; Goldberg, 1991; Goldberg & Gloger, 1992) or require special type information to be passed to functions at runtime (Tolmach, 1994). By requiring values in certain regions to be of the same kind, our approach has much in common with BIBOP (Big Bag Of Pages) systems, with regions as pages (Hanson, 1980).

Another particular body of related work investigates the notion of escape analysis for improving stack allocation in garbage collected systems (Blanchet, 1998; Salagnac *et al.*, 2005). Region inference and MLKit's polymorphic multiplicity analysis (Birkedal *et al.*, 1996) allow more objects to be stack allocated than traditional escape analyses, which allows only local, non-escaping values to be stack allocated. Other work investigates the use of static prediction techniques and linear typing for inferring heap space usage (Jost *et al.*, 2010).

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. 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 & Aiken, 2001). A modern language for system programming is Rust, which is based on ownership types

[6] The scheme works well together with support for unboxed data constructors, such as cons (`::`), which, for instance, leads to a compact representation of linked lists (Elsman, 1998).

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

for controlling the use of resources, including memory (Aldrich *et al.*, 2002). Ownership types are also used for real-time implementations of Java (Boyapati *et al.*, 2003). None of the above systems are combined with techniques for automatic generational garbage collection.

Also related to the present work is the work by Aiken *et al.* (1995), who show how region inference may be improved for some programs by removing the constraints of the stack discipline, which may cause a garbage collector to run less often. Other work in this area includes (Fluet *et al.*, 2006), which removes the constraints of the region stack discipline for an intermediate language using a linear type system.

Region inference has also been used in practical settings without combining it with reference-tracing garbage collection. In particular, it has been used as the primary memory management scheme for a web server (Elsman & Hallenberg, 2003; Elsman *et al.*, 2018).

7 Conclusion and future work

We have presented a technique for combining region inference and generational garbage collection in a functional language. Whereas region inference alone can be beneficial when a program is optimised for regions, in general, a combination with reference-tracing garbage collection is necessary to make region inference perform well in practice.

Whereas region inference significantly reduces the number of garbage collections needed, and therefore also reduces the accumulated garbage collection time, combining region inference with generational garbage collection is shown to decrease the accumulated garbage collection time even further. The combination of region inference and generational garbage collection is therefore found to be superior to the combination that involves non-generational garbage collection, despite the overhead introduced by a larger amount of memory waste, due to region fragmentation.

In relation to the performance differences between MLton and MLKit, a first obvious candidate for future work is to improve the x64 code generator and to apply some of the datatype specialisation techniques that are implemented in the MLton compiler. Second, for making the combination of region inference and generational garbage collection useful for applications that make heavy use of mutable objects, a proper implementation of a "remembered set" would be an appropriate next step. Finally, an obvious candidate for future work is to investigate the possibility of combining region inference and, perhaps, generations, with features for concurrency and parallelism.

Acknowledgments

We are grateful to the anonymous referees for their many helpful comments on earlier drafts of this paper.

Conflicts of interest

None.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

References

Aditya, S., Flood, C. H. & Hicks, J. E. (1994) Garbage collection for strongly-typed languages using run-time type reconstruction. In *LISP and Functional Programming,* pp. 12–23.

Aiken, A., Fähndrich, M. & Levien, R. (1995) Better static memory management: Improving region-based analysis of higher-order languages. In *ACM Conference on Programming Languages and Implementation*. PLDI 1995.

Aldrich, J., Kostadinov, V. & Chambers, C. (2002) Alias annotations for program understanding. In *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA 2002.

Anderson, T. A. (2010) Optimizations in a private nursery-based garbage collector. In *ACM International Symposium on Memory Management*. ISMM 2010.

Appel, A. W. (1989) Runtime tags aren't necessary. *Lisp Sym. Comput.* **2**, 153–162.

Birkedal, L., Tofte, M. & Vejlstrup, M. (1996) From region inference to von Neumann machines via region representation inference. In *ACM Symposium on Principles of Programming Languages*. POPL 1996.

Blanchet, B. (1998) Escape analysis : Correctness proof, implementation and experimental results. *ACM Symposium on Principles of Programming Languages (POPL 1998)*. ACM Press, pp. 25–37.

Boyapati, C., Salcianu, A., Beebee, Jr., W. & Rinard, M. (2003) Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation*. PLDI 2003.

Calcagno, C., Helsen, S. & Thiemann, P. (2002) Syntactic type soundness results for the region calculus. *Inform. Comput.* **173**(2).

Doligez, D. & Leroy, X. (1993) A concurrent, generational garbage collector for a multi-threaded implementation of ML. In *ACM Symposium on Principles of Programming Languages*. POPL '93.

Elsman, M. (1998) Polymorphic equality—no tags required. In *Second International Workshop on Types in Compilation*.

Elsman, M. (2003) Garbage collection safety for region-based memory management. In *ACM Workshop on Types in Language Design and Implementation*. TLDI 2003.

Elsman, M. (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.

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

Elsman, M. & Hallenberg, N. (2003) Web programming with SMLserver. In *International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*. Springer-Verlag.

Elsman, M. & Hallenberg, N. (2020) On the effects of integrating region-based memory management and generational garbage collection in ML. In *Practical Aspects of Declarative Languages*. PADL 2020. Springer International Publishing, pp. 95–112.

Elsman, M., Munksgaard, P. & Larsen, K. F. (2018) Experience report: Type-safe multi-tier programming with Standard ML modules. In *Proceedings of the ML Family Workshop*. ML 2018.

Fluet, M., Morrisett, G. & Ahmed, A. (2006) Linear regions are all you need. *Program. Lang. Syst.* ESOP 2006. Springer Berlin Heidelberg, pp. 7–21.

Gay, D. & Aiken, A. (2001) Language support for regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*. ACM Press.

Goldberg, B. (1991) Tag-free garbage collection for strongly typed programming languages. In *ACM Conference on Programming Language Design and Implementation* pp. 165–176.

Goldberg, B. & Gloger, M. (1992) Polymorphic type reconstruction for garbage collection without tags. In *LISP and Functional Programming* pp. 53–65.

Hallenberg, N. (1996) *A Region Profiler for a Standard ML compiler based on Region Inference*. Student Project 96-5-7, Department of Computer Science, University of Copenhagen (DIKU).

Hallenberg, N., Elsman, M. & Tofte, M. (2002) Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation (PLDI 2002)*. ACM Press. Berlin, Germany.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press

Hanson, D. R. (1980) A portable storage management system for the icon programming language. *Softw. Pract. Exp.* **10**, 489–500.

Helsen, S. & Thiemann, P. (2000) Syntactic type soundness for the region calculus. In *International Workshop on Higher Order Operational Techniques in Semantics*. Published in Volume 41(3) of the Electronic Notes in Theoretical Computer Science.

Huelsbergen, L. & Winterbottom, P. (1998) Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *ACM International Symposium on Memory Management*. ISMM 1998.

Jones, R., Hosking, A. & Moss, E. (2011) *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC.

Jost, S., Hammond, K., Loidl, H.-W. & Hofmann, M. (2010) Static determination of quantitative resource usage for higher-order programs. In *ACM Symposium on Principles of Programming Languages*. POPL 2010.

Marlow, S. & Peyton Jones, S. (2011) Multicore garbage collection with local heaps. In *ACM International Symposium on Memory Management*. ISMM 2011.

Marlow, S., Peyton Jones, S. & Singh, S. (2009) Runtime support for multicore Haskell. In *ACM International Conference on Functional Programming*. ICFP 2009.

Morrisett, G. (1995) *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Morrisett, G., Felleisen, M. & Harper, R. (1995) Abstract models of memory management. In *International Conference on Functional Programming Languages and Computer Architecture*, San Diego, pp. 66–77.

Morrisett, G., Tarditi, D., Cheng, P., Stone, C., Harper, R. & Lee, P. (1996) *The TIL/ML Compiler: Performance and Safety through Types*.

Reppy, J. H. (1994) *A High-performance Garbage Collector for Standard ML*. Tech. rept. AT&T Bell Laboratories.

Salagnac, G., Yovine, S. & Garbervetsky, D. (2005) Fast escape analysis for region-based memory management. *Electron. Notes Th. C. S.* **131**(May), 99–110.

Salagnac, G., Nakhli, C., Rippert, C. & Yovine, S. (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*.

Swamy, N., Hicks, M., Morrisett, G., Grossman, D. & Jim, T. (2006) Safe manual memory management in cyclone. *Sci. Comput. Program.* **62**(2), 122–144.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R. & Lee, P. (1996) TIL: A type-directed optimizing compiler for ML. In *Proceedings of ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation,* pp. 181–192.

Tofte, M. & Birkedal, L. (1998) A region inference algorithm. *Trans. Program. Lang. Syst. (TOPLAS)* **20**(4), 734–767.

Tofte, M. & Birkedal, L. (2000) Unification and polymorphism in region inference. In *Proof, Language, and Interaction. Essays in Honour of Robin Milner* May. (25 pages).

Tofte, M. & Talpin, J.-P. (1993) *A Theory of Stack Allocation in Polymorphically Typed Languages*. Tech. rept. DIKU-report 93/15. Department of Computer Science, University of Copenhagen.

Tofte, M. & Talpin, J.-P. (1997) Region-based memory management. *Inform. Comput.* **132**(2), 109–176.

Tofte, M., Birkedal, L., Elsman, M. & Hallenberg, N. (2004) A retrospective on region-based memory management. *Higher-Order Symb. Comput.* **17**(3), 245–265.

Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T. H. & Sestoft, P. (2006) *Programming with Regions in the MLKit (Revised for Version 4.3.0)*. Tech. Rept. IT University of Copenhagen, Denmark.

Tolmach, A. P. (1994) Tag-free garbage collection using explicit type parameters. In *LISP and Functional Programming* pp. 1–11.

Ueno, K. & Ohori, A. (2016) A fully concurrent garbage collector for functional programs on multicore processors. In *ACM International Conference on Functional Programming*. ICFP 2016.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inform. Comput.* **115**(1), 38–94.

https://doi.org/10.1017/S0956796821000010 Published online by Cambridge University Press