A Concurrent Language with a Uniform Treatment of Regions and Locks

Prodromos Gerakios Nikolaos Papaspyrou Konstantinos Sagonas
School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{pgerakios,nickie,kostis}@softlab.ntua.gr

Abstract

A challenge for programming language research is to design and implement multi-threaded low-level languages providing static guarantees for memory safety and freedom from data races. Towards this goal, we sketch a concurrent language employing safe region-based memory management and hierarchical locking of regions. Both regions and locks are treated uniformly, and the language supports ownership transfer, early deallocation of regions and early release of locks in a safe manner.

1 Introduction

Writing safe and robust code is a hard task; writing safe and robust multi-threaded low-level code is even harder. In this paper we sketch a minimal, low-level concurrent language with advanced region-based memory management and hierarchical lock-based synchronization primitives.

Region-based memory management achieves efficiency by bulk allocation and deallocation of objects in segments of memory called regions. Similar to other approaches, our regions are organized in a hierarchical manner such that each region is physically allocated within a single parent region and may contain multiple children regions. This hierarchical structure imposes an ownership relation as well as lifetime constraints over regions. Unlike other languages employing hierarchical regions, our language allows early subtree deallocation in the presence of region sharing between threads. In addition, no memory leaks are possible as each thread is obliged to release each region it owns by the end of its scope.

Multi-threaded programs that interact through shared memory generate random execution interleavings. A data race occurs in a multi-threaded program when there exists an interleaving such that some thread accesses a memory location while some other thread attempts to write to it. So far, type systems and analyses that guarantee race freedom [5] have mainly focused on lexically-scoped constructs. The key idea in those systems is to statically track or infer the lockset held at each program point. In the language presented in this paper, implicit re-entrant locks are used to protect regions from data races. Our locking primitives are non-lexically scoped. Locks also follow the hierarchical structure of regions so that each region is protected by its own lock as well as the locks of all its ancestors, in contrast with ownership types [2].

Furthermore, our language allows regions and locks to be safely aliased, escape the lexical scope when passed to a new thread, or become logically separated from the remaining hierarchy. These features are invaluable for expressing numerous idioms of multi-threaded programming such as *sharing*, *region ownership* or *lock ownership transfers*, *thread-local regions* and *region migration*.

2 Language Design

We briefly outline the main design goals for our language, as well as some of the main design decisions that we made to serve these goals.

Low-level and concurrent. Our language must efficiently support systems programming. As such, it should cater for memory management and concurrency. It also needs to be low-level: it is not intended to be used by programmers but as the target language of higher-level systems programming languages.

Static safety guarantees. We define safety in terms of *memory safety* and absence of *data races*. A static type system should guarantee that well-typed programs are safe, with minimal run-time overhead.

Safe region-based memory management. Similarly to other languages for safe systems programming (e.g. Cyclone) our language employs region-based memory management, which achieves efficiency by *bulk allocation* and *deallocation* of objects in segments of memory (*regions*). Statically typed regions [11, 12] guarantee the absence of dangling pointer dereferences, multiple release operations of the same memory area, and memory leaks. Traditional stack-based regions [11] are limiting as they cannot be deallocated early. Furthermore, the stack-based discipline fails to model region lifetimes in concurrent languages, where the lifetime of a shared region depends on the lifetime of the longest-lived thread accessing that region. In contrast, we want regions that can be *deallocated early* and that can safely be *shared* between concurrent threads.

We opt for a *hierarchical region* [7] organization: each region is physically allocated within a single parent region and may contain multiple children regions. Early region deallocation in our multi-level hierarchy automatically deallocates the immediate subtree of a region without having to deallocate each region of the subtree recursively. The hierarchical region structure imposes the constraint that a child region is *accessible* only when its ancestors are accessible. In order to allow a function to access a region without having to pass all its ancestors explicitly, we allow its ancestors to be abstracted (i.e., our language supports *hierarchy abstraction*) for the duration of the function call. To maintain the *accessibility* invariant we require that the abstracted parents are *accessible* before and after the call. Regions whose parent information has been abstracted cannot be passed to a new thread as this may be unsound.

Race freedom. To prevent data races we use *lock-based* mutual exclusion. Instead of having a separate mechanism for locks, we opt for a uniform treatment of locks and regions: locks are placed in the same hierarchy as regions and enjoy similar properties. Each region is protected by its own private lock and by the locks of its ancestors. The semantics of region locking is that the entire subtree of a region is *atomically locked* once the lock for that region has been acquired. Hierarchical locking can model complex synchronization strategies and lifts the burden of having to deal with explicit acquisition of multiple locks. Although deadlocks are possible, they can be *avoided* by acquiring a single lock for a group of regions rather than acquiring multiple locks for each region separately. Additionally, our language provides explicit locking primitives, which in turn allow a higher degree of concurrency than lexically-scoped locking, as some locks can be released early.

Region polymorphism and aliasing. Our language supports *region polymorphism*: it is possible to pass regions as parameters to functions or concurrent threads. This enables *region aliasing*: one actual region could be passed in the place of two distinct formal region parameters. In the presence of mutual exclusion and early region deallocation, aliasing is dangerous. Our language allows safe region aliasing with minimal restrictions. The mechanism that we employ for this purpose also allows us to encode numerous useful idioms of concurrent programming, such as *region migration*, *lock ownership transfers*, *region sharing*, and *thread-local regions*.

3 Language Description

The syntax of the language is illustrated in Figure 1. We only present the subset of the language that is related to regions and references. The language core comprises variables (x), constants (c), functions, and function application. Functions can be region polymorphic ($\Lambda\rho.\, f$) and region application is explicit ($e[\rho]$). Monomorphic functions ($\lambda x.\, e$) must be annotated with their type. The application of monomorphic functions is annotated with a *calling mode* (ξ), which is seq for normal (sequential) application and $\mathsf{par}(\epsilon)$ for spawning a new thread (parallel). Parallel application is annotated with the list of regions (ϵ) that migrate to the spawned thread. This annotation can be automatically inferred by the type checker. The constructs for manipulating references are standard. A newly allocated memory cell is returned by

Function	f	::=	$\lambda x.e$ as $\tau \xrightarrow{\gamma\to\gamma} \tau \mid \Lambda\rho.f$
Expression	e	::=	$x \mid c \mid f \mid (e\ e)^{\xi} \mid e[\rho] \mid$ new e at $e \mid e := e$
		$\mid$	deref $e \mid$ newrgn ρ, x at e in $e \mid$ cap$_\eta$ e
Type	τ	::=	$b \mid \tau \xrightarrow{\gamma\to\gamma} \tau \mid \forall\rho.\tau \mid$ ref$(\tau,\rho) \mid$ rgn(ρ)
Effect	γ	::=	$\emptyset \mid \gamma, \rho^{\kappa} \triangleright \pi$

Calling mode	ξ	::=	seq $\mid$ par(ϵ)
Capability op	η	::=	$\psi+ \mid \psi-$
Capability kind	ψ	::=	rg $\mid$ lk
Capability	κ	::=	$n,n \mid \overline{n,n}$
Region parent	π	::=	$\rho \mid \bot \mid ?$
Region list	ϵ	::=	$\emptyset \mid \epsilon,\rho$

Figure 1: Syntax.

new e_1 at e_2, where e_1 is the value that will be placed in the cell and e_2 is a handle of the region in which the new cell will be allocated. Standard assignment and dereference operators complete the picture.

The construct newrgn ρ, x at e_1 in e_2 allocates a new region ρ with a region *handle* bound to x. The new region resides in a *parent* region, whose handle is given in e_1. The scope of ρ and x is e_2, which is supposed to do something useful with the new region. The evaluation of e_2 is *obliged to consume* the new region by the end of its scope. A region can be consumed either by deallocation or by transferring its ownership to another thread. At any given program point, each region is associated with a *capability* (κ). Capabilities consist of two natural numbers, the *capability counts*: the *region* count, which denotes whether the region is live, and the *lock* count, which denotes whether the region has been locked to provide the current thread with exclusive access to its contents. When first allocated, a region starts with capability $(1,1)$, meaning that it is live and locked for providing exclusive access to the thread which allocated it (this our equivalent of a thread-local region).

By using the construct cap$_\eta$ e, a thread can *increment* or *decrement* the capability counts of the region whose handle is specified in e. The capability operator η can be, e.g., rg+ (meaning that the region count is to be incremented) or lk− (meaning that the lock count is to be decremented). If the region count reaches zero, then the region may be physically deallocated and no subsequent operation on it can be performed. If the lock count becomes zero, then the region is unlocked. Capability counts determine the validity of operations on regions and references. All operations require that the involved regions are live, i.e., the region count is greater than zero. Assignment and dereference can be performed only when the corresponding region is live and locked.

A capability of the form (n_1,n_2) is called a *pure* capability, whereas a capability of the form $(\overline{n_1,n_2})$ is called an *impure* capability. In both cases, it is implied that the current thread can decrement the region count n_1 times and the lock count n_2 times. Impure capabilities are obtained by splitting, in the same spirit as *fractional capabilities* [4], pure or other impure capabilities into several pieces, e.g., $(3,2) = (\overline{2,1}) + (\overline{1,1})$. These pieces are useful for region aliasing, when the same region is to be passed to a function in the place of two distinct region parameters. An impure capability implies that our knowledge of the region and lock count is inexact. The use of such capabilities must be restricted; e.g., an impure capability with a non-zero lock count cannot be passed to another thread, as it is unsound to allow two threads to simultaneously hold the same lock. Capability splitting takes place automatically with function application.

4 Static Semantics

In this section we discuss the most interesting parts of our type system. To enforce our safety invariants, we use a *type and effect system*. Effects are used to statically track the capability of each region. An effect (γ) is a list of elements of the form $\rho^{\kappa} \triangleright \pi$, denoting that region ρ is associated with capability κ and has parent π, which can be another region, $\bot$, or ?. Regions whose parents are $\bot$ or ? are considered as roots in our region hierarchy. We assume that there is an initial (physical) root region corresponding to

$$\frac{\Delta \vdash \tau \qquad \tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \qquad \Delta;\Gamma, x:\tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2)}{\Delta;\Gamma \vdash \lambda x.e \text{ as } \tau : \tau \,\&\, (\gamma;\gamma)} \text{ (T-F)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \,\&\, (\gamma;\gamma') \qquad \Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma';\gamma'') \qquad \xi = \mathsf{par}(\epsilon) \Rightarrow \tau_2 = \langle\rangle \qquad \xi \vdash \gamma''' = \gamma_2 \oplus (\gamma'' \ominus \gamma_1)}{\Delta;\Gamma \vdash (e_1\ e_2)^{\xi} : \tau_2 \,\&\, (\gamma;\gamma''')} \text{ (T-AP)}$$

$$\frac{\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,\rho) \,\&\, (\gamma;\gamma') \qquad \rho \in \mathit{accessible}(\gamma')}{\Delta;\Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, (\gamma;\gamma')} \text{ (T-D)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(\rho') \,\&\, (\gamma;\gamma') \qquad \rho' \in \mathit{dom}(\gamma') \qquad \Delta \vdash \tau \qquad \Delta,\rho;\Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma', \rho^{1,1} \triangleright \rho';\gamma'') \qquad \rho \notin \mathit{dom}(\gamma'')}{\Delta;\Gamma \vdash \mathtt{newrgn}\ \rho, x\ \mathtt{at}\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma'')} \text{ (T-NG)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \qquad \Delta;\Gamma \vdash e_2 : \mathtt{rgn}(\rho) \,\&\, (\gamma';\gamma'') \qquad \rho \in \mathit{dom}(\gamma'')}{\Delta;\Gamma \vdash \mathtt{new}\ e_1\ \mathtt{at}\ e_2 : \mathtt{ref}(\tau,\rho) \,\&\, (\gamma;\gamma'')} \text{ (T-NR)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(\rho) \,\&\, (\gamma;\gamma', \rho^{\kappa} \triangleright \pi) \qquad \kappa' = [\![\eta]\!](\kappa) \qquad \gamma'' = \mathit{live}(\gamma', \rho^{\kappa'} \triangleright \pi)}{\Delta;\Gamma \vdash \mathtt{cap}_{\eta}\ e_1 : \langle\rangle \,\&\, (\gamma;\gamma'')} \text{ (T-CP)}$$

Figure 2: Selected typing rules.

$$\frac{\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \qquad \gamma'' = \mathit{live}(\gamma') \qquad \mathit{consistent}(\gamma,\gamma'') \qquad \xi = \mathsf{seq} \Rightarrow \mathit{abs\_par}(\gamma,\gamma_1) \subseteq \mathit{dom}(\gamma'') \qquad \xi = \mathsf{par}(\epsilon) \Rightarrow \gamma_2 = \emptyset \wedge \epsilon = \mathit{pure}(\gamma) \cap \mathit{dom}(\gamma_1)}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)} \text{ (ESJ)}$$

$$\frac{}{\xi \vdash \gamma = \emptyset \oplus \gamma} \text{ (ES-N)} \qquad \frac{\pi' \in \{\pi, ?\} \qquad \xi = \mathsf{par}(\epsilon) \Rightarrow \pi' \neq ? \qquad \xi \vdash \kappa = \kappa_1 + \kappa_2 \qquad \xi \vdash \gamma = \gamma_1 \oplus \gamma_2}{\xi \vdash \gamma, r^{\kappa} \triangleright \pi = \gamma_1, r^{\kappa_1} \triangleright \pi' \oplus \gamma_2, r^{\kappa_2} \triangleright \pi} \text{ (ES-C)}$$

$$\frac{rg(\kappa) = rg(\kappa_1) + rg(\kappa_2) \qquad lk(\kappa) = lk(\kappa_1) + lk(\kappa_2) \qquad rg(\kappa_1) > 0 \qquad \mathit{is\_pure}(\kappa_1) \Leftrightarrow \mathit{is\_pure}(\kappa_2) \qquad \mathit{is\_pure}(\kappa_1) \Rightarrow \kappa = \kappa_1 \qquad \xi \neq \mathsf{seq} \wedge \neg \mathit{is\_pure}(\kappa_1) \Rightarrow lk(\kappa_2) = 0}{\xi \vdash \kappa = \kappa_1 + \kappa_2} \text{ (CS)}$$

Figure 3: Effect and capability splitting.

the entire heap, whose handle is available to the main program. The parent of the heap region is $\bot$. More (logical) root regions can be created using hierarchy abstraction. The abstract parent of a region that is passed to a function is denoted by ?.

The syntax of types in Figure 1 (on page 3) is more or less standard. A collection of base types b is assumed; the syntax of values belonging to these types and operations upon such values is omitted from this paper. We assume the existence of a *unit* base type, which we denote by $\langle\rangle$. Region handle types $\mathtt{rgn}(\rho)$ and reference types $\mathtt{ref}(\tau,\rho)$ are associated with a type-level region name ρ. Monomorphic function types carry an *input* and an *output effect*. A well-typed expression e has a type τ under an input effect γ and results in an output effect γ'. We denote this by $\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$. The typing relation (cf. Figure 2) uses two standard typing contexts: Δ, a set of region variables, and Γ, a mapping of term variables to types. The effects that appear in our typing relation must satisfy a *liveness invariant*: all regions that appear in the effect are *live*, i.e. their region counts and those of all their ancestors are non-zero. Thus, in order to check whether a region ρ is live in the current effect γ, we only need to check that $\rho \in \mathit{dom}(\gamma)$.

The typing rule for lambda abstraction (*T-F*) requires that the body e is well-typed with respect to the effects ascribed on its type. The typing rule for function application (*T-AP*) splits the output effect of e_2 (γ'') by subtracting the function's input effect (γ_1). It then joins the remaining effect with the function's output effect (γ_2). In the case of parallel application, rule *T-AP* also requires that the return type is unit. The splitting and joining of effects is controlled by the judgement $\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$, which is defined in Figure 3 (the auxiliary functions and predicates are defined in Figures 4 and 5). Its definition enforces the following properties:

- the liveness invariant for γ'';
- the consistency of γ and γ'', i.e. regions cannot change parent and capabilities cannot switch from pure to impure or vice versa;

$$\frac{(\rho^{\kappa} \triangleright \pi) \in \gamma \qquad \pi \in \{\bot, ?\}}{\mathit{is\_live}(\gamma, \rho)} \qquad \frac{(\rho^{\kappa} \triangleright \rho') \in \gamma \qquad \mathit{is\_live}(\gamma, \rho')}{\mathit{is\_live}(\gamma, \rho)}$$

$$\frac{(\rho^{\kappa} \triangleright \pi) \in \gamma \qquad \mathit{lk}(\kappa) > 0}{\mathit{is\_accessible}(\gamma, \rho)} \qquad \frac{(\rho^{\kappa} \triangleright \rho') \in \gamma \qquad \mathit{is\_accessible}(\gamma, \rho')}{\mathit{is\_accessible}(\gamma, \rho)}$$

Figure 4: Auxiliary predicates: region liveness and accessibility.

$$\begin{array}{lcl}
\mathit{rg}(\kappa) & = & n_1 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\mathit{lk}(\kappa) & = & n_2 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\mathit{dom}(\gamma) & = & \{\rho \mid (\rho^{\kappa} \triangleright \pi) \in \gamma\} \\
\mathit{live}(\gamma) & = & \{\rho^{\kappa} \triangleright \pi \mid (\rho^{\kappa} \triangleright \pi) \in \gamma \wedge \mathit{is_live}(\gamma, \rho)\} \\
\mathit{accessible}(\gamma) & = & \{\rho \mid (\rho^{\kappa} \triangleright \pi) \in \gamma \wedge \mathit{is_accessible}(\gamma, \rho)\} \\
\mathit{is_pure}(\kappa) & = & \exists n_1.\, \exists n_2.\, \kappa = n_1, n_2 \\
\mathit{pure}(\gamma) & = & \{r^{\kappa} \triangleright \pi \mid (r^{\kappa} \triangleright \pi) \in \gamma \wedge \mathit{is_pure}(\kappa)\} \\
\mathit{consistent}(\gamma_1, \gamma_2) & = & \forall (\rho^{\kappa} \triangleright \pi) \in \gamma_1.\, \forall (\rho^{\kappa'} \triangleright \pi') \in \gamma_2.\, \pi = \pi' \wedge (\mathit{is_pure}(\kappa) \Leftrightarrow \mathit{is_pure}(\kappa')) \\
\mathit{abs_par}(\gamma_1, \gamma_2) & = & \left\{\rho \mid (\rho^{\kappa} \triangleright \rho') \in \gamma_1 \wedge (\rho^{\kappa'} \triangleright ?) \in \gamma_2\right\}
\end{array}$$

Figure 5: Auxiliary functions and predicates.

- in the case of sequential application, regions that pass under hierarchy abstraction must be live after the function returns;
- in the case of parallel application, the thread output effect is empty, the thread input effect does not contain impure capabilities with non-zero lock counts, no hierarchy abstraction is permitted, and regions that migrate are exactly those given by the annotation ϵ.

The typing rules for references are standard. Here we only show the rules for dereference (*T-D*) and reference allocation (*T-NR*). The former checks that the region ρ where the reference resides is *accessible*, i.e. ρ itself or one of its ancestors has a non-zero lock count in the current effect. The latter just checks that the region ρ is live. The rule for creating new regions (*T-NG*) first checks that e_1 is a handle for some live region ρ'. The body expression e_2 is type checked in a context extended with the fresh region ρ and its handle x. A new element $\rho^{1,1} \triangleright \rho'$ is added to the input effect of e_2, stating that the new region is thread-local (live, locked and not known to anybody else). The rule also checks that the type of the result τ and the output effect γ'' do not contain any occurrence of region variable ρ. This implies that e_2 must have consumed the new region by the end of its scope. The capability manipulation rule (*T-CP*) checks that the given expression is the handle of a live region ρ. It then modifies the capability count of the effect element corresponding to that region, as dictated by function $[\![\eta]\!]$ which increases or decreases the region or the lock count of its argument, according to the value of η. The dynamic semantics makes sure that evaluation returns only when the actual capability counts are consistent with the desired output effect. For instance, if the lock of region ρ is held by some other executing thread, evaluation of $\mathtt{cap}_{\mathsf{lk}+}$ must be suspended until the lock can be obtained. On the other hand, evaluation of $\mathtt{cap}_{\mathsf{rg}-}$ does not need to suspend but may not be able to physically deallocate a region, as it may be in use by other threads.

5 Related Work

The first statically checked stack-based region system was developed by Tofte and Talpin [11]. Since then, several memory-safe systems that enabled early region deallocation for a sequential language were proposed [1, 10, 13, 6]. Cyclone [9] and RC [7] were the first imperative languages to allow safe region-based management with explicit constructs. They both allowed early region deallocation and RC also

introduced the notion of multi-level region hierarchies. RC programs may throw region-related exceptions, whereas our approach is purely static. Both Cyclone and RC make no claims of memory safety or race freedom for concurrent programs. In the context of Cyclone, Grossman proposed a type system for safe multi-threading [8]. Race freedom is guaranteed by statically tracking locksets within lexically-scoped synchronization constructs. Grossman's proposal allows for fine-grained locking, but only deals with stack-based regions and does not enable early release of regions and locks. Furthermore, we offer hierarchical locking as opposed to just primitive locking. Bulk region deallocation is impossible in Grossman's system.

Statically checked region systems have also been proposed [3, 15, 14] for real-time Java to rule out dynamic checks imposed by its specification. Boyapati et al. [3] introduce hierarchical regions in ownership types but the approach suffers from the same disadvantages as Grossman's work. Additionally, their type system only allows sub-regions for *shared* regions, whereas we do not have this limitation. Boyapati also proposed an ownership-based type system that prevents deadlocks and data races [2]. Static region hierarchies (depth-wise) have been used by Zhao [15]. Their main advantage is that programs require fewer annotations compared to programs with explicit region constructs. In the same track, Zhao et al. [14] proposed implicit ownership annotations for regions. Thus, classes that have no explicit owner can be allocated in any static region. This is a form of *existential ownership*. In contrast, we allow a region to completely abstract its owner/ancestor information by using the *hierarchy abstraction* mechanism. None of the above approaches allow full ownership abstraction for region subtrees.

The main limitation of our work is that we require explicit annotations regarding ownership and region capabilities. Moreover, our locking system offers coarser-grained locking compared to [5] and related type systems. The use of hierarchical locking avoids some, though not all, deadlocks.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, New York, NY, USA, June 1995. ACM Press.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, New York, NY, USA, Nov. 2002. ACM Press.

[3] C. Boyapati, A. Salcianu, W. S. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 324–337, New York, NY, USA, June 2003. ACM Press.

[4] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, June 2003.

[5] C. Flanagan and M. Abadi. Object types against races. In J. C. M. Baeten and S. Mauw, editors, *Concurrency Theory: Proceedings of the 10th International Conference*, volume 1664 of *LNCS*, pages 288–303. Springer, 1999.

[6] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In P. Sestoft, editor, *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 3924 of *LNCS*, pages 7–21. Springer, Mar. 2006.

[7] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, New York, NY, USA, May 2001. ACM Press.

[8] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 13–25, New York, NY, USA, Jan. 2003. ACM Press.

[9] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, New York, NY, USA, June 2002. ACM Press.

[10] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 175–186, New York, NY, USA, 2001. ACM.

[11] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, New York, NY, USA, Jan. 1994. ACM Press.

[12] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst.*, 22(4):701–771, July 2000.

[13] D. Walker and K. Watkins. On regions and linear types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 181–192, New York, NY, USA, Oct. 2001. ACM Press.

[14] T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. Implicit ownership types for memory management. *Sci. Comput. Program.*, 71(3):213–241, 2008.

[15] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251. IEEE Computer Society, 2004.