Race-free and Memory-safe Multithreading: Design and Implementation in Cyclone

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

We present the design of a formal low-level multi-threaded language with advanced region-based memory management and synchronization primitives, where well-typed programs are memory safe and race free. In our language, regions and locks are combined in a single hierarchy and are subject to uniform ownership constraints imposed by a hierarchical structure: deallocating a region causes its sub-regions to be deallocated. Similarly, when a region is protected, then its sub-regions are also protected. We discuss aspects of the integration and implementation of the formal language within Cyclone and evaluate the performance of code produced by the modified Cyclone compiler against highly optimized C programs using atomic operations, pthreads, and OpenMP. Although our implementation is still in a preliminary stage, our results show that the performance overhead for guaranteed race freedom and memory safety is acceptable.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages; D.1.3 [*Software*]: Concurrent Programming—Parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Design, Languages, Theory

Keywords Safe multithreading, type and effect systems, region-based memory management, Cyclone

1. Introduction

With the emergence of commodity multicore architectures, exploiting the performance benefits of multi-threaded execution has become increasingly important to the extent that doing so is arguably a necessity these days. Programming languages that retain the transparency and control of memory such as C, seem best-suited to exploit the performance benefits of multicore machines, except for the fact that programs written in them often compromise memory safety by allowing invalid memory accesses, buffer overruns, space leaks, etc. and become susceptible to data races by careless uses of locks. Thus, a challenge for programming language research is to design and implement multithreaded low-level languages providing static guarantees for memory safety and freedom from data races and at the same time allow for a relatively smooth conversion of legacy C code to its multi-threaded counterpart.

Towards this challenge, we present the design of a formal low-level concurrent language that employs advanced region-based management and hierarchical lock-based synchronization primitives. Similar to other approaches, our memory regions are organized in a hierarchical manner where each region is physically allocated within a single parent region and may contain multiple children regions. Our language allows deallocation of complete subtrees in the presence of region sharing between threads and deallocation is allowed to occur at any program point. Each region is associated with an implicit lock. Thus, locks also follow the hierarchical structure of regions and in this setting each region is protected by its own lock as well as the locks of all its ancestors. As opposed to the majority of type systems and analyses that guarantee race freedom for lexically-scoped locking constructs [8, 12, 21], our language employs non-lexically scoped locking primitives, which are more suitable for languages at the C level of abstraction. Furthermore, the formal language allows regions and locks to be safely aliased and escape the lexical scope when passed to a new thread. These features are invaluable for expressing numerous idioms of multi-threaded programming such as *sharing*, *region ownership* or *lock ownership transfers*, and *region migration*.

More importantly, our formal language is not just a theoretical design with some nice properties. We have integrated our language in Cyclone [22], a strongly-typed dialect of C which preserves explicit control and representation of memory without sacrificing memory soundness. We have opted for Cyclone both because it has a publicly available implementation but also because it is more than a safe variant of C. Cyclone offers modern programming language features such as first-class polymorphism, exceptions, tuples, namespaces, (extensible) algebraic data types, and region-based memory management. We will discuss how these features interact with our language and the additions that were needed to Cyclone's implementation.

The contributions of this paper are as follows:

- We improve on our earlier work [18] by providing an operational semantics for our language that not only provides explicit guarantees for race freedom and memory safety, but also explicit guarantees as to when a subtree is deallocated, thereby avoiding temporary memory leaks.
- We discuss the integration of our formal language to Cyclone. The resulting language is a concurrent language at the C level of abstraction that enjoys the benefits of the formal system: it offers memory safety and race freedom guarantees and allows regions to be deallocated in bulk and also be locked atomically.

TLDI'10, January 23, 2010, Madrid, Spain.

- We discuss implementation issues related to analysis, code generation and additions to the run-time system that were required in order to make the integration possible.

The next section reviews the Cyclone language. We present our language and its operational semantics in Section 3. We describe the interaction of our language with Cyclone in Section 4, followed by a presentation of implementation (Section 5) and performance (Section 6) aspects of its integration. The paper ends by three short sections presenting future improvements to our implementation, discussing related work, and some concluding remarks.

2. Cyclone: A memory-safe dialect of C

In this section, we provide a high-level overview of memory management in Cyclone. In particular, we discuss how traditional regions are used in Cyclone and identify some shortcomings that are alleviated in our language. Additionally, we clarify through several examples that Cyclone's memory safety guarantees only hold for sequential programs.

2.1 Memory management in Cyclone

Cyclone employs a uniform treatment of different memory segments such as the main heap, the stack and individual regions. More specifically, memory segments are mapped into *logical memory partitions*. Each data object is allocated at a single memory segment, but references to objects may refer to multiple segments. Hereon, we overload the term "region" to mean a type-level logical memory partition, a run-time entity that enables fast allocation and bulk deallocation of objects, or a memory segment such as the heap and the stack.

For instance, a stack frame is treated as a region holding the values of variables declared in a lexical block. As another example, the main heap is an immortal region that contains all global variables. The type system of Cyclone tracks the set of *live* regions at each program point and verifies that the regions associated with each accessed object are indeed a subset of the live regions.

```
{ region < `r > h ;          // live regions:    {`r}
  int * `r z = rnew(h) 10;   //                  {`r}
      ...                    //                  {`r}
}                            //                  { }
```

The above example illustrates how a scoped region can be created and used: the first statement allocates a fresh memory segment, and associates this segment with a fresh type-level region (i.e., $`r$). Following Cyclone's terminology, we use a leading backquote for type-level names, e.g. $`r$. (We will often use the same name without the backquote for the corresponding region handle, which is here explicitly named h.) The comments on the right-hand side of the example's code show the live region set (i.e., the *effect*) at each program point.

The new region can be accessed via its *region handle* (h), which is given the *singleton type* `region_t` $< `r >$. The second statement uses $h$ to allocate memory for a single integer and initializes it to the value 10. The type of the fresh reference is annotated with region $`r$ (i.e., `int` $* `r$). The type system ensures that the reference can only be accessed when $`r$ is in the current effect.

The uniform treatment of memory allows for polymorphism over different kinds of memory segments.

```
void swap (int * `r1 x, int * `r2 y );
```

For instance, the above line of code defines a function that swaps the contents of the variables x and y located at regions $`r_1$ and $`r_2$ respectively. Both $`r_1$ and $`r_2$ are polymorphic and can be instantiated with any region. The following line of code invokes *swap* by explicitly instantiating both $`r_1$ and $`r_2$ to $`r$.

```
{ region < `r > h ;
  int * `r z = rnew(h) 10;
  int * `r y = rnew(h) 15;
      swap(z, y);              // effect of swap   {`r, `r}
}
```

As shown in the above example, type-level regions can be freely aliased in a effect (e.g. $\{`r, `r\}$). The downside of this approach is that scoped regions can only be deallocated implicitly by the run-time system when a region's scope ends.

To overcome this restriction, the authors of Cyclone have extended the region system with three powerful features, namely *tracked types*, the notion of *borrowing* tracked types and *existential types*. Tracked types, which are closely related to linear types, disallow aliasing of *tracked* references. Borrowing can be used to convert a tracked reference to an aliasable reference within a particular scope. The aliasable reference is accessible within the scope, whereas the tracked reference becomes inaccessible for the duration of the scope. Finally, existential types serve as the means for overcoming lexically scoped region names, by permitting the on-demand concealment and disclosure of region names. Cyclone allows access and deallocation of non-lexically scoped (i.e., *dynamically* scoped) regions as follows:

- A request is made to the run-time system to allocate a fresh dynamic region.
- The run-time system returns an existential package containing some region name $`r$ and a *key* (i.e., a tracked reference) to the handle of the fresh region. The handle is also annotated with $`r$.
- The existential package is unpacked and $`r$ is brought into scope as well as the key.
- The program can immediately deallocate the new region by deallocating the key,
- or it may temporarily yield access to the key by allowing it to be *borrowed* within a scope. When this happens, $`r$ is added to the effect and the region referred by the key is usable.

The following example illustrates a similar scenario:

```
void access_and_deallocate (NewDynRgn pr ) {
  let NewDynRgn{< `r > key} = pr;     // open existential
  {region h = open(key);              // borrow key for this scope
   let x = rnew(h) 5;
   ...
  }
  free_ukey(key);                     // deallocate region
}
```

It should be noted that a dynamic region cannot be deallocated when its key has been *borrowed*. Additionally, Cyclone allows tracked references to leak and thus allows dynamic regions to leak as well. To tackle this issue, an intra-procedural analysis can be used to report tracked reference leaks. In practice, this analysis is impractical as it produces a large number of false positives [28]. For instance, when a function call takes place between the allocation and deallocation point of a tracked reference, the analysis must report that the tracked reference may leak as an uncaught exception may be thrown during the call. For a detailed discussion about memory management aspects of Cyclone we refer the reader to the work of Swamy *et al.* [28].

As it will be discussed in the following sections, our work disallows memory leaks in the presence of a complex shared memory management scheme with bulk region deallocation, allows region deallocation at *any program point* and simplifies the process of creating, using and deallocating explicitly freeable regions.

2.2 Concurrency in Cyclone

Cyclone does not have language support for concurrency. Instead, it provides an interface to the pthreads library, which allows programmers to spawn new threads and use numerous synchronization primitives to control the interaction between threads. The interface to the pthreads library ensures that the run-time data structures are correctly initialized before a new thread runs.

To preserve memory safety (e.g., absence of dangling pointers), Cyclone requires that all memory regions passed to a new thread must live at least as long as the immortal (main) heap. This implies that threads can interact with other threads via dynamically allocated references that reside in the heap or in global variables. This restriction diminishes the explicit memory management benefits of Cyclone (aliasable heap references can only be garbage collected). The following definition has been extracted from Cyclone's interface to pthreads library:

```
int pthread_create(pthread_t @, const pthread_attr_t *,
                   'a(@'H)('b), 'b arg : regions('b) ≤ 'H)
```

The most interesting part of the above definition is $\texttt{regions}(\text{`}b) \leq \text{`}H$, which says that all region names occurring in the type that will instantiate the type variable $\text{`}b$ must be live at least as long as the immortal heap ($\text{`}H$). Tracked pointers cannot be passed to threads.

The memory safety guarantees of Cyclone can be compromised in the presence of multi-threading. Here, we only mention a few cases which can violate memory safety. Firstly, the data flow analysis performed for identifying where dynamic checks (e.g., null pointer and array bounds checks) should be inserted is unsound in a concurrent setting. Consider the following code fragment:

```
void foo(int * 'r * 'r x) {
    if (x != NULL && *x != NULL) **x = 20;
}
```

Assuming that x is a shared *possibly null* reference, then the analysis will deduce that $**x$ can be accessed within the conditional statement as x and $*x$ are definitely *not null*. This property does not hold for concurrent programs that share x, but do not synchronize their accesses to it.

Secondly, some features of Cyclone such as pattern matching, accesses to wide references (i.e., *fat pointers*) and *swap* operations between tracked references must be performed *atomically*. The lack of atomicity in swap operations and wide references can trivially compromise memory safety and cause dangling pointer dereferences and double "free" operations.

Last but not least, Cyclone's type system does not guard against data races. The absence of data races gives additional guarantees to the programmer and allows a thread-aware compiler to perform certain kinds of optimizations that should only be applied to sequential programs. As it will be shown in the sections that follow, we have solved some of the issues stated above by implementing an adjusted version of our type system and operational semantics in Cyclone. We have also re-engineered the run-time system so that it is mostly non-blocking and thread-safe. The next section introduces our type system and operational semantics.

3. Formal language

Our earlier work on a hierarchical region type system provides an operational semantics that satisfies the design goals for our language, but admits *temporary leaks* of regions and locks [18]. The memory leaks are temporary as the type system enforces that all regions and locks will eventually be released and can be avoided, by disallowing the release of *aliased* regions. However, this restriction would impede the expressiveness and the benefits of our language, as region aliasing is the rule, not the exception.

Function	f	::=	$\lambda x.\, e \;\texttt{as}\; \tau \xrightarrow{\gamma\to\gamma} \tau \mid \Lambda\rho.\, f$
Expression	e	::=	$x \mid \texttt{c} \mid f \mid (e\ e)^{\xi} \mid e[r] \mid \texttt{new}\ e\ \texttt{at}\ e$
		|	$e := e \mid \texttt{deref}\ e \mid \texttt{newrgn}^{r}\ \rho, x\ \texttt{at}\ e\ \texttt{in}\ e$
		|	$\texttt{cap}^{r}_{\eta}\ e \mid \texttt{rgn}_{\iota} \mid \texttt{loc}_{l} \mid \texttt{pop}_{\gamma}\ e$
Type	τ	::=	$b \mid \langle\rangle \mid \tau \xrightarrow{\gamma\to\gamma} \tau \mid \forall\rho.\, \tau \mid \texttt{ref}(\tau, r) \mid \texttt{rgn}(r)$
Effect	γ	::=	$\emptyset \mid \gamma, r^{\kappa} \triangleright \pi$
Capability kind	ψ	::=	rg | lk
Capability op	η	::=	$\psi+ \mid \psi-$
Region	r	::=	$\rho \mid \iota \mid \iota@n$
Capability	κ	::=	$n, n \mid \overline{n, n}$
Region parent	π	::=	$r \mid \bot$
Calling mode	ξ	::=	seq | par

Figure 1. Syntax.

Here, we improve on earlier work, by providing a safe semantics for the same type system that disallows leaks. The key idea is the introduction of dynamic effects and the preservation of an exact correspondence between dynamic and static effects.

3.1 Language description

The language syntax is illustrated in Figure 1.[1] The core expressions include variables (x), constants (c), functions, and function application. Function application terms are annotated with a *calling mode* (ξ). The calling mode specifies whether a function application should be executed sequentially (seq) or in parallel (par). Monomorphic functions ($\lambda x.\, e$) must be annotated with their type (τ). Our language also includes region-polymorphic functions ($\Lambda\rho.\, f$) and region application ($e[\rho]$). Other constructs can be easily included, as long as conservative choices are made to ensure the soundness of the type system (e.g., in the standard `if-then-else` construct, both branches should produce the same effect).

The construct $\texttt{newrgn}^{r}\ \rho, x\ \texttt{at}\ e_1\ \texttt{in}\ e_2$ allocates a fresh region ρ at the region indicated by handle e_1, and binds x to the *handle* of ρ. Both ρ and x are lexically bound to the scope of e_2. The new region must be explicitly released within e_2. The region allocation construct is annotated with the parent region name r, which is only required for the type safety proof.

The constructs for manipulating references are standard. A newly allocated memory cell is returned by $\texttt{new}\ e_1\ \texttt{at}\ e_2$, where e_1 is an initializer expression for the new cell and handle e_2 indicates the region in which the new cell will be allocated. Standard assignment and dereference operators complete the picture. A region can be released 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 and *lock* count, which denote whether a region is live and locked respectively. When first allocated, a region starts with capability $(1, 1)$, meaning that it is live and locked, so that it can be accessed directly with no additional overhead. This is our equivalent of a thread-local region.

By using the construct $\texttt{cap}^{r}_{\eta}\ e$, a thread can *increment* or *decrement* the capability counts of some region r whose handle is specified in e. The annotation (r) on the `cap` construct is only required

[1] The constructs $\texttt{rgn}_{\iota}$, $\texttt{loc}_{\text{l}}$ and $\texttt{pop}_{\gamma}\ e$ are not considered part of the language. They are only introduced during program evaluation. We defer the discussion about them until Section 3.2.

Stack	σ	$::= \emptyset \mid \sigma;\gamma$
Hierarchy	δ	$::= \emptyset \mid \delta, n \mapsto \sigma$
Contents	H	$::= \emptyset \mid H, \ell \mapsto v$
Region list	S	$::= \emptyset \mid S, \iota \mapsto H$
Threads	T	$::= \emptyset \mid T, n : e$
Configuration	C	$::= \delta; S; T$

$$E ::= \square \mid (E\ e)^{\xi} \mid (v\ E)^{\xi} \mid E[r] \mid \texttt{newrgn}^{r}\,\rho, x\,\texttt{at}\,E\,\texttt{in}\,e \mid \texttt{cap}^{r}_{\eta}\,E$$
$$\mid \texttt{new}\,v\,\texttt{at}\,E \mid \texttt{deref}\,E \mid E := e \mid v := E \mid \texttt{new}\,E\,\texttt{at}\,e \mid \texttt{pop}_{\gamma}\,E$$

Figure 2. Configuration, store, threads and evaluation contexts.

for the type safety proof. 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). Incrementing counts is essential for sharing regions among threads and for region aliasing. Furthermore, incrementing a lock count from 0 to 1 amounts to acquiring a region lock, which may have to block the current thread if the lock is held by another thread. On the other hand, decrementing counts amounts to releasing capabilities. When a region count reaches zero, no subsequent operations can be performed on this region and the region may be physically deallocated (if no other threads are using it). When a lock count reaches zero, the region is unlocked, but it may still be *protected* by a locked ancestor region. As we explained, capability counts determine the validity of operations on regions and references. All memory-related 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 protected.

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 pure or other impure capabilities into several pieces, e.g., the pure capability $(3, 2)$ can be split into two impure capabilities $\overline{(2, 1)}$ and $\overline{(1, 1)}$, in the same spirit as *fractional capabilities* [10]. Splitting a linear resource into multiple pieces is particularly useful for region aliasing (e.g., the same region can 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 counts held by the current thread is inexact. Under certain circumstances, the use of impure capabilities must be disallowed; 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 access the same region. Capability splitting takes place automatically with function application.

3.2 Operational semantics

We define a *small-step* operational semantics for our language, using two evaluation relations, at the level of *threads* and *expressions* (Figures 3 and 4 on the next page). The thread evaluation relation transforms *configurations*. A configuration C (see Figure 2) consists of global hierarchy δ, an abstract *store* S and a thread map T.[2] The global hierarchy δ maps thread identifiers (n) to stacks (σ). A thread stack σ is a list of frames (γ) and represents a hierarchy of regions accessible to a thread. Each frame γ represents the portion of σ that is accessible to the function that is currently executed. Notice, that frames include region counts. A frame is a list of elements of the form $r^{\kappa} \triangleright \pi$, denoting that region r is associated with count κ and has parent π, which can be another region or $\bot$. Regions whose parents are $\bot$ are considered as roots in a region hierarchy. A store S maps region identifiers (ι) to heaps (H). A heap H, maps memory locations to values. A thread map T associates thread identifiers to expressions (i.e., threads).

A *thread evaluation context* E (Figure 2) is defined as an expression with a *hole*, represented as $\square$. The hole indicates the position where the next reduction step can take place. Our notion of evaluation context imposes a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order.

We assume that concurrent reduction events can be totally ordered [25]. At each step, a random thread (n) is chosen from the thread list for evaluation (Figure 3). It should be noted that the thread evaluation rules are the only *non-deterministic* rules in the operational semantics of our language; in the presence of more than one active thread, our semantics does not specify which one will be selected for evaluation. Threads that have completed their evaluation, have released all regions used by them, and have been reduced to *unit* values, represented as (), are removed from the active thread list (rule *E-T*). Rule *E-S* reduces some thread n via the expression evaluation relation. Notice, that rule *E-S* only modifies the stack of thread n and requires that the resulting hierarchy δ' is *consistent* ($\vdash \delta'$): regions accessible to thread n should be inaccessible to other threads and regions having positive pure capabilities can only be live at a *single* stack frame of thread n.[3] Therefore, the operational semantics will get stuck if the mutual exclusion protocol is unsatisfied. Our approach differs from related work, e.g. the work of Grossman [21], where a special kind of value $junk_v$ is often used as an intermediate step when assigning a value v to a location, before the real assignment takes place, and type safety guarantees that no junk values are ever read.

When a parallel function application redex is detected within the evaluation context of a thread, a new thread is created (rule *E-SN*). The redex is replaced with a unit value in the currently executed thread and a new thread is added to the thread list, with a *fresh* thread identifier. The calling mode of the application term is changed from parallel to sequential. The topmost frame of the spawning thread (γ) is split into two frames γ' and γ_1 so that the intersection of regions locked in γ' and in γ_1 is empty.[4] If it is impossible to split γ, the thread evaluation relation gets stuck. Notice, that γ_1 is an effect annotation of the function abstraction. Frame γ is then replaced by γ' and γ_1 becomes the initial frame of the new thread.

The expression evaluation relation (defined in Figure 4) rewrites tuples of the form $\sigma; S; e$, where σ is a thread local stack, S is the global store, and e is an expression. Constant regions may be of the form $\iota@n$, which is a constant region ι tagged with a unique identifier n. The region application (*E-RP*) rule introduces tags during substitution so as to prevent the existence of duplicate region names in function effects.

Hereon, the symbol γ means "the topmost frame of the currently executed thread n". The sequential function application (*E-A*) rule splits γ into two stack frames γ_1 and γ_r such that γ_1 matches the effect expected by the lambda abstraction, and substitutes the sequence of stack frames $\gamma_r; \gamma_1$ for γ. The function body is placed within a pop construct, which is annotated with frame γ_r. A pop construct must not be contained in the original program, and must

[2] The order of elements in comma-separated lists, e.g. in a store S or in a list of threads T, is unimportant; we consider all list permutations as equivalent.

[3] The second invariant ensures that regions with positive pure capabilities can safely be passed to other threads (e.g., locked). This is sound when the current thread has no more counts of such regions in other stack frames.

[4] The rules for splitting effects are defined in Figure 6 and discussed in Section 3.3. Until then, the judgement $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$ should be interpreted as saying that the effect γ' is what we get if we start with γ, remove the input effect γ_1 of the function that we are calling and (when the function returns) add the function's output effect γ_2.

$$\frac{v_1 \equiv \lambda x.\, e \;\texttt{as}\; \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \qquad \delta = \delta'', n \mapsto \sigma; \gamma \qquad \text{fresh } n' \qquad \mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1) \qquad \delta' = \delta'', n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1}{\delta; S; T, n : E[(v_1\ v)^{\mathsf{par}}] \rightsquigarrow \delta'; S; T, n : E[()], n' : (v_1\ v)^{\mathsf{seq}}} \quad \textit{(E-SN)}$$

$$\frac{\delta = \delta'', n \mapsto \sigma \qquad \sigma; S; e \to \sigma'; S'; e' \qquad \delta' = \delta'', n \mapsto \sigma' \qquad \vdash \delta'}{\delta; S; T, n : E[e] \rightsquigarrow \delta'; S'; T, n : E[e']} \quad \textit{(E-S)} \qquad \frac{\delta = \delta', n \mapsto (\emptyset; \emptyset)}{\delta; S; T, n : () \rightsquigarrow \delta'; S; T} \quad \textit{(E-T)}$$

Figure 3. Thread evaluation relation $C \rightsquigarrow C'$.

$$\frac{\sigma = \sigma'; \gamma \qquad \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \sigma'' = \sigma'; \gamma_r; \gamma_1}{\sigma; S; ((\lambda x.\, e \;\texttt{as}\; \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2)\ v)^{\mathsf{seq}} \to \sigma''; S; \mathtt{pop}_{\gamma_r}\ e[v/x]} \quad \textit{(E-A)}$$

$$\frac{\sigma = \sigma'; \gamma_r; \gamma' \qquad \mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma_r \ominus \emptyset) \qquad \sigma'' = \sigma'; \gamma''}{\sigma; S; \mathtt{pop}_{\gamma_r}\ v \to \sigma''; S; v} \quad \textit{(E-E)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_live}(\gamma, r) \qquad \text{fresh } \iota \qquad \sigma'' = \sigma'; \gamma, \iota^{1,1} \triangleright r}{\sigma; S; \mathtt{newrgn}^r\ \rho, x\ \mathtt{at}\ \mathtt{rgn}_{\bar{r}}\ \mathtt{in}\ e \to \sigma''; S, \iota \mapsto \emptyset; e[\iota/\rho][\mathtt{rgn}_\iota/x]} \quad \textit{(E-NG)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_live}(\gamma, r) \qquad \gamma = \gamma', r^{\kappa} \triangleright \pi \qquad \kappa' = [\![\eta]\!](\kappa) \qquad \sigma'' = \sigma'; \text{live}(\gamma', r^{\kappa'} \triangleright \pi)}{\sigma; S; \mathtt{cap}^r_\eta\ \mathtt{rgn}_{\bar{r}} \to \sigma''; S; ()} \quad \textit{(E-C)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_live}(\gamma, r) \qquad \text{fresh } \ell}{\sigma; S; \mathtt{new}\ v\ \mathtt{at}\ \mathtt{rgn}_{\bar{r}} \to \sigma; S[\bar{r} \mapsto S(\bar{r}), \ell \mapsto v]; \mathtt{loc}_\ell} \quad \textit{(E-NR)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_accessible}(\gamma, r) \qquad (\ell \mapsto v_1) \in S(\bar{r})}{\sigma; S; \mathtt{loc}_\ell := v \to \sigma; S(\bar{r})[\ell \mapsto v]; ()} \quad \textit{(E-AS)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_accessible}(\gamma, r) \qquad (\ell \mapsto v) \in S(\bar{r})}{\sigma; S; \mathtt{deref}\ \mathtt{loc}_\ell \to \sigma; S; v} \quad \textit{(E-D)} \qquad \frac{\text{fresh } n'}{\sigma; S; (\Lambda\rho.\, f)[r] \to \sigma; S; f[\bar{r}@n'/\rho]} \quad \textit{(E-RP)}$$

Figure 4. Expression evaluation relation $\sigma; S; e \to \sigma'; S'; e'$.

only appear during program evaluation. Rule *E-E* eliminates `pop` constructs, when the function body has been reduced to a value and the annotation γ_r of `pop` matches the frame preceding the topmost frame γ'. Frames γ_r are γ' are joined to form a new frame γ'', which replaces them on the current stack.

The remaining rules of Figure 4 make use of the judgements *is_live*(γ, r) and *is_accessible*(γ, r) (Figure 7) to establish that a region r is *live* and *accessible* in a frame γ. A region r is *live* in γ when the region count of each region in the path between r and the root region is positive. A region r is *accessible* in γ when it is live and there exists at least one region in the path between r and the root region with a positive lock count. We also define the following partial functions: $\bar{r}$ removes the unique identifier from a tagged region, $[\![\eta]\!](\kappa)$ decrements or increments the region or lock field of κ by one, according to operation specified by η, and finally live(γ) selects a subset of γ so that all regions in that subset are live.

Rule *E-NG* requires that region r is live in γ, adds a fresh and empty region ι to S and adds the dynamic effect of ι to γ, which specifies that r is the parent of ι and that ι has region and lock count of one. Rule *E-C* requires that region r is live in γ, substitutes $[\![\eta]\!](\kappa)$ for κ in γ at the exponent of r, and removes dead regions from the resulting frame. Rule *E-NR* requires that region r is live in γ and updates the heap of r with a fresh location ℓ mapping to value v. Notice, that r may be *unlocked*. Rules *E-AS* and *E-D* require that *some region* r, which contains the location (ℓ) being accessed, must be *accessible* in γ. Therefore, the semantics will get stuck when a thread attempts to access a memory location without having acquired an appropriate lock for this location.

3.3 Static semantics

We discuss the most interesting aspects of our type system. We employ a *type and effect system* to enforce memory and race safety invariants. Effects (γ) are used to statically track region capabilities.

The syntax of types has been defined in Figure 1. A collection of base types b is assumed; the syntax of values belonging to these types and operations upon such values are omitted from this paper. We assume the existence of a *unit* base type, which we denote by $\langle\rangle$. Region handle types $\mathtt{rgn}(r)$ and reference types $\mathtt{ref}(\tau, r)$ are associated with a type-level region r. 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'. The typing relation (see Figure 5) is denoted by $R; M; \Delta; \Gamma \vdash e : \tau\ \&\ (\gamma; \gamma')$ and uses four typing contexts: a set of region literals (R), a mapping of locations to types (M), a set of region variables (Δ), and a mapping of term variables to types (Γ).

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 by rule *ESJ* in Figure 6. It uses the simpler judgement $\xi \vdash \gamma = \gamma_1 \oplus \gamma_2$ for just splitting (not joining) effects, which is defined by rules *ES-N* and *ES-C*. This, in turn, uses a third judgement $\xi \vdash \kappa = \kappa_1 \oplus \kappa_2$ for splitting capabilities, which is defined by rule *CS*. Some auxiliary functions and predicates are defined in Figure 7.

As defined in Figure 6 the rules for splitting and joining effects enforce the following properties:

- The effect after the join satisfies the liveness invariant, i.e., all regions that appear in it are live (their region counts and those of all their ancestors are positive). This is enforced by $\gamma'' = \mathit{live}(\gamma')$ in rule *ESJ*.
- For parallel application, the thread output effect must be empty. In other words, every thread is obliged to deallocate all its regions and release the locks that it holds.
- Regions with pure capabilities cannot appear more than once in a function's input or output effect. In other words, region aliasing is only allowed with impure capabilities. This is enforced by $ok(\gamma_1; \gamma_2)$ in rule *ESJ*.
- Capability and effect splitting is not symmetric. In $\xi \vdash \kappa = \kappa_1 \oplus \kappa_2$, capability κ_1 goes to the function being called and capability κ_2 is what "stays behind" (the same is true for effects). Under this light, the rule *CS* ensures three things:
  - Capabilities that stay behind are of the same purity as the original ones. This implies that capabilities do not change purity as a result of splitting and joining.

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

$$\frac{R;M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \quad \mathit{is\_accessible}(\gamma', r)}{R;M;\Delta;\Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, (\gamma;\gamma')} \ \textit{(T-D)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma') \quad \mathit{is\_live}(\gamma', r) \quad R;\Delta \vdash \tau \quad R;M;\Delta,\rho;\Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma', \rho^{1,1} \triangleright r;\gamma'') \quad \rho \notin \mathit{dom}(\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{newrgn}^{r}\ \rho, x\ \mathtt{at}\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma'')} \ \textit{(T-NG)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \quad \mathit{is\_live}(\gamma'', r) \quad R;M;\Delta;\Gamma \vdash e_2 : \mathtt{rgn}(r) \,\&\, (\gamma';\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{new}\ e_1\ \mathtt{at}\ e_2 : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma'')} \ \textit{(T-NR)}$$

$$\frac{\mathit{is\_live}(\gamma', r^{\kappa} \triangleright \pi, r) \quad \gamma'' = \mathit{live}(\gamma', r^{\kappa'} \triangleright \pi) \quad R;M;\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma', r^{\kappa} \triangleright \pi) \quad \kappa' = [\![\eta]\!](\kappa)}{R;M;\Delta;\Gamma \vdash \mathtt{cap}^{r}_{\eta}\ e_1 : \langle\rangle \,\&\, (\gamma;\gamma'')} \ \textit{(T-CP)}$$

Figure 5. Selected typing rules.

$$\frac{\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \quad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \quad \gamma'' = \mathit{live}(\gamma') \quad \mathit{ok}(\gamma_1;\gamma_2) \quad \xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)} \ \textit{(ESJ)}$$

$$\frac{}{\xi \vdash \gamma = \emptyset \oplus \gamma} \ \textit{(ES-N)}$$

$$\frac{\xi \vdash \gamma, r^{\kappa_2} \triangleright \pi = \gamma_1 \oplus \gamma_2 \quad \xi \vdash \kappa = \kappa_1 + \kappa_2 \quad \pi \simeq \pi' \quad r' \simeq r}{\xi \vdash \gamma, r^{\kappa} \triangleright \pi = \gamma_1, r'^{\kappa_1} \triangleright \pi' \oplus \gamma_2} \ \textit{(ES-C)}$$

$$\frac{\mathit{rg}(\kappa) = \mathit{rg}(\kappa_1) + \mathit{rg}(\kappa_2) \quad \mathit{lk}(\kappa) = \mathit{lk}(\kappa_1) + \mathit{lk}(\kappa_2) \quad \mathit{is\_pure}(\kappa) \Leftrightarrow \mathit{is\_pure}(\kappa_2) \quad \mathit{is\_pure}(\kappa_1) \Rightarrow \kappa = \kappa_1 \quad \xi = \mathsf{par} \wedge \neg \mathit{is\_pure}(\kappa_1) \Rightarrow \mathit{lk}(\kappa_1) = 0}{\xi \vdash \kappa = \kappa_1 + \kappa_2} \ \textit{(CS)}$$

Figure 6. Effect and capability splitting.

$$\frac{(r^{\kappa} \triangleright \bot) \in \gamma \quad \mathit{rg}(\kappa) > 0}{\mathit{is\_live}(\gamma, r)} \qquad \frac{\gamma = \gamma', r^{\kappa} \triangleright r' \quad \mathit{rg}(\kappa) > 0 \quad \mathit{is\_live}(\gamma', r')}{\mathit{is\_live}(\gamma, r)}$$

$$\frac{(r^{\kappa} \triangleright \pi) \in \gamma \quad \mathit{lk}(\kappa) > 0 \quad \mathit{is\_live}(\gamma, r)}{\mathit{is\_accessible}(\gamma, r)} \qquad \frac{\gamma = \gamma', r^{\kappa} \triangleright r' \quad \mathit{lk}(\kappa) = 0 \quad \mathit{rg}(\kappa) > 0 \quad \mathit{is\_accessible}(\gamma', r')}{\mathit{is\_accessible}(\gamma, r)}$$

Figure 7. Auxiliary predicates: region liveness and accessibility.

- If a pure capability is passed to a function, nothing stays behind.
- If an impure capability is passed to a spawned thread, the lock count must be zero.

It is interesting to notice that the pure capability $(2, 1)$ can be split as $(\overline{1, 0}) \oplus (1, 1)$, but not as $(1, 1) \oplus (\overline{1, 0})$. In the case of parallel application, it cannot be split as $(\overline{1, 1}) \oplus (1, 0)$ either.

- In the effect that is passed to the called function, regions (and their parents) need not be identical to those in the original effect. Rule *ES-C* only requires that $r \simeq r'$ (and $\pi \simeq \pi'$), which means that equality is checked after erasing the region tags that are introduced by the operational semantics (i.e., replacing $\iota@n$ with ι). This requirement is only important for the proof of type safety.

The typing rules for references are standard. In Figure 5 we only show the rules for dereference (*T-D*) and reference allocation (*T-NR*). The former checks that region r is *accessible*. The latter only checks that the region r is *live*. Notice that effect typing is left-to-right, which is consistent with the left-to-right evaluation in the operational semantics. The output effect of the rightmost subexpression of each construct is always (except for rule *T-NG*) used for checking the liveness and accessibility invariants. The rule for creating new regions (*T-NG*) checks that e_1 is a handle for some live region r'. Expression e_2 is type checked in an extended typing context (i.e., ρ and $x : \mathtt{rgn}(\rho)$ are appended to Δ and Γ respectively) and an extended input effect (i.e., a new effect is appended to the input effect such that the new region is live and accessible to this thread). The rule also checks that the type and the output effect of e_2 do not contain any occurrence of region variable ρ. This implies that ρ must be *consumed* by the end of the scope of e_2.

The capability manipulation rule (*T-CP*) checks that e is a handle of a live region r. It then modifies the capability count of r 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 ensures that an operational step is performed if the updated hierarchy preserves the invariant that protected regions are accessible to a single thread at instance of time. For instance, if the lock of region r is held by some other executing thread, the evaluation of $\mathtt{cap}_{\mathsf{lk}+}$ must be suspended until the lock can be obtained. On the other hand, the 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 used by other threads.

Type safety is based on proving the *preservation* and *progress* lemmata.[5] Deadlocked threads are not considered to be stuck. A well-typed configuration $\delta; S; T$ is *not stuck* when each thread in T can take one of the evaluation steps in Figure 3 (*E-S*, *E-T* or *E-SN*) or when it is waiting for a lock held by some other thread. Given these definitions, the *progress* and *preservation* lemmata are standard. The type safety theorem can be formulated as follows:

Theorem 1 (Type safety) *Let T_0 be the program's initial thread list, containing only the* `main` *thread with identifier* 1. *Let δ_0 and S_0 be the program's initial hierarchy and region list, such that there is only the heap region ι_H, with capability (1, 0) for thread* 1. *If the operational semantics takes any number of steps $\delta_0; S_0; T_0 \rightsquigarrow^* \delta; S; T$, then the resulting configuration $\delta; S; T$ is not stuck.*

[5] Full proofs and a full formalization of our language are given in the companion technical report [19].

4. Interaction with Cyclone

We have integrated the type system and operational semantics presented in earlier sections to Cyclone. In this section, we provide an in-depth description of the interaction between our system and Cyclone. We have identified five important goals that the integration should accomplish or preserve:

- Memory safety: dangling and null pointer dereferences as well as buffer overruns should be prevented.
- Thread safety: the type system should guarantee that shared data accesses should be race free. Furthermore, the run-time system must be re-entrant and thread safe. We defer the discussion about implementation issues until the next section.
- Separate compilation: it should be possible to compile and link separate modules independently.
- Backwards compatibility: sequential Cyclone code should work as expected with no further modifications.
- Accessing local data without synchronization: thread-local data is often the rule, not the exception. It should be possible to access thread-local data with no additional overhead (i.e. without synchronization).

4.1 Extended regions

In contrast with traditional lexically scoped regions, which are allocated in a LIFO manner, our *extended* regions can be allocated at any extended region ancestor. We consider the main heap ($`H$) as the root of our region hierarchy. Thus, the *heap_handle* can also be used for allocating an extended region.

The following example illustrates this point. In particular, we allocate a fresh extended region ($`child$) within an existing region $`parent$. The type system ensures that $`parent$ is *live* at the allocation point, but not necessarily *accessible*.

```
{ region child @ parent   // add `child^{1,1}▷ `parent to effect
  ...
  xdec(child);            // remove `child from effect
  ...
}
```

As in traditional regions, *child* is the handle to the fresh region. This form of allocation generalizes the stack-based region organization to a tree-based organization and enables finer-grained control of region lifetimes.

As in the operational semantics, region $`child$ is a *sharable* region, but no synchronization is required for accessing its data, as it is initially *accessible* to the thread allocating it.

4.2 Traditional Cyclone effects

The *effect* system of Cyclone tracks the set of *accessible* regions at each program point. Functions are annotated with a *single* effect, which can be automatically inferred by calculating the union of region variables occuring in the types of function parameters. As mentioned, traditional regions cannot be deallocated once they have been added in a function's effect, as unrestricted region aliasing within an effect is admitted. Therefore, a function effect serves as both a precondition and a postcondition of the regions that are accessible before and after calling a function respectively.

The following example illustractes a function, which has been annotated explicitly with the effect $\{`r\}$. This effect implies that region $`r$ is both *live* and *accessible* for the entire scope of foo.

```
void foo ( region_t < `r > h ; {`r});
```

We have decided to place our effects in separate annotations as full effect inference for our regions is beyond the scope of this work and we wish to preseve backwards compatibility with traditional Cyclone programs that enjoy full inference.

Our effects are mutually exclusive with traditional effects so a region name may not exist in both effects. We expect that future implementations will integrate the two different kinds of effects into a single effect and will enjoy effect inference for both traditional and extended regions.

The following example shows how we can write function *foo* so that it uses extended regions:

```
void foo ( region_t < `r > h ) @ieffect({`r, i(1,1), `H})
                               @oeffect({`r, i(1,1), `H});
```

The `@ieffect` and `@oeffect` annotations denote the input and output effects of function *foo*. These effects consist solely of extended regions or $`H$.[6] The equivalent type of function *foo* in our formal type system would be:

$$\forall `r.\, \texttt{region\_t} < `r > \xrightarrow{\gamma\to\gamma} \langle\rangle \quad \text{where } \gamma \equiv `r^{\overline{1,1}} \triangleright \bot$$

The heap region is mapped to $\bot$ as it is immortal. Impure capabilities $\overline{n_1, n_2}$ are denoted by $\texttt{i}(n_1, n_2)$, whereas pure capabilities n_1, n_2 are denoted by $\texttt{p}(n_1, n_2)$. Pure capabilities are most useful when transferring lock capabilites to other threads. It is therefore expected that impure capabilities would be the common case. Therefore, the above definition can be abbreviated as follows:

```
void foo( region_t < `r > h ) @ieffect({`r, 1, 1, `H})
                              @oeffect({`r, 1, 1, `H});
```

Finally, it is possible to omit the output effect annotation when a function consumes the regions declared at the input effect.

4.3 Hierarchy abstraction

In order to allow a function to access a region without having to pass all its ancestors explicitly, its ancestors can be abstracted from an effect for the duration of a function call. To maintain soundness, we require that abstracted parents are *live* before and after the call. Regions whose parent information has been abstracted cannot be passed to a new thread as this may be unsound. The definition of *foo* can be further simplified, by using hierarchy abstraction:

```
void foo ( region_t < `r > h ) @ieffect({`r, 1, 1})
                               @oeffect({`r, 1, 1});
```

4.4 Operating on capabilities

The `cap` operator of the formal semantics has been encoded as a set of library functions:

```
void xdec ( region_t <`r>; )  @ieffect({`r, 1, 0});
void xinc  ( region_t <`r>; )  @ieffect({`r, 1, 0})
                               @oeffect({`r, 2, 0});
void xldec( region_t <`r>; )  @ieffect({`r, 1, 1})
                               @oeffect({`r, 1, 0});
void xlinc (region_t <`r>; )  @ieffect({`r, 1, 0})
                               @oeffect({`r, 1, 1});
```

For instance, the first function *xdec* encodes the operator $\texttt{cap}_{\texttt{rg}-}$ for any region $`r$. It requires that the calling context has at least one region capability. This invariant is encoded in its input effect. The output effect of *xdec* is empty, thus exactly one region capability is consumed. Similary the remaning functions encode the remaining functionality of operator `cap`. It would be preferable to use dependent types to allow these functions to increment or decrement counts by more than one. To the best of our knowledge this

[6] $`H$ can only occur as a parent annotation.

is impossible to express at the type level, in Cyclone's type system. However, we plan on extending the type-level expressiveness in future versions.

4.5 Exceptions

Having static guarantees about the control flow of a program plays a crucial role in manual memory management. As mentioned in earlier sections, Cyclone allows memory leaks[7] of tracked objects (e.g., dynamic regions).

We decided that our regions should *always* be reclaimed manually. Towards this goal, we have made possible to annotate Cyclone function declarations with uncaught exception names that may be thrown from a function's body.[8] We have not opted for an interprocedural analysis as this would disallow separate compilation.

In addition, exact knowledge of a function's control-flow graph is required to guarantee soundness: if the body of a function does not satisfy the `@oeffect` postcondition of that function (as a result of a statically unknown exception), then it is possible to introduce dangling pointers. There exist three kinds of annotations for exceptions:

- the `@throws(...)` clause enumerates all exceptions that may be thrown from a function body.
- the `@nothrow` annotation is an abbreviation for `@throws()`.
- finally, `@throwsany` acts as a wildcard for any exception that may be thrown. This annotation may be useful for legacy library prototypes and code.

The default annotation for functions is `@throwsany`. Exceptions may be thrown *explicitly* by the programmer or *implicitly* by the run-time system. Implicit exceptions arise in situations where:

- a *null* pointer is dereferenced.
- an *out of bounds* array access is performed.
- the run-time system has *insufficent memory* to fulfill an allocation request.
- a value *cannot be matched* against any of the available patterns.

The exception analysis takes into consideration both explicit and implicit exceptions. In the future we plan on relaxing the exception analysis and adding run-time support to ensure that function postconditions are always satisfied.

4.6 Re-entrant functions

Global data is implicitly shared by all threads and this may cause a data race. To preserve race-freedom, we have constrained our language so that only extended regions can be shared between threads. (Although we allow reading global variables that are declared as constant.) Traditional regions (or references) cannot be passed to threads.

To enforce, this policy we require that each explictly spawned thread must be declared as `@re_entrant`. A function annotated as `@re_entrant` yields access to global variables, the immortal heap, tracked objects and it can only invoke `@re_entrant` functions. Function *main*, is not `@re_entrant`. Global data and tracked objects can still be directly accessed by any non re-entrant function invoked directly or indirectly by function *main*. Therefore, sequential programs have full access to global data. In the future, we intend to relax type-checking so that tracked objects can be passed to threads, provided that these objects are consumed from the environment performing a spawn operation.

[7] These pointers will eventually be reclaimed by the garbage collector.

[8] We have noticed that Cyclone has a `@throws` clause but it is undocumented and not functioning.

4.7 Thread creation

Threads can be explicitly created by the means of the `spawn` operator. This operator takes two expressions e_1 and e_2, i.e., `spawn` (e_1) e_2, and spawns a new thread. The first expression is a list of thread-specific parameters such as the stack size. The second expression e_2 must be a function call and the function must be annotated as `@re_entrant@nothrow` and its `@oeffect` annotation must be either empty or omitted. Furthermore, the traditional Cyclone effect must be *empty* so that unsharable regions cannot be used in the new thread. Both expressions e_1 and e_2 are evaluated from left to right. The spawning thread does not block and returns immediately.

4.8 Type polymorphism

Cyclone effects are not polymorphic. To allow the invocation of functions, which have polymorphic arguments (e.g., say ‘a), Cyclone programmers use the `regions`(‘a) operator. The purpose of this operator is to defer effect checking until the function call is performed, where the calling environment must prove that all regions occuring in the type that instantiates ‘a are present in the environment's effect:

`void` *foo* (‘a ; `regions`(‘a));

In terms of extended regions, the `regions` operator would require that all regions occuring in ‘a are *live* and *accessible* for the scope of the function call. However, this is beyond the scope of our type system. Furthermore, we cannot provably guarantee memory safety if this construct is used in the way described above. Therefore, the type checker disallows uses of `regions` operator within our effects (e.g. `@ieffect`).

This limitation could be improved in future work but as a workaround we allow extended regions to interoperate with traditional regions. We explain this feature in the following subsection.

4.9 Interoperability with traditional regions

The distinction between traditional and extended regions may be limiting for programs that require both kinds of regions. We introduce a language construct similar to the `alias` and `open` constructs of Cyclone, that borrows a part (or a fraction) of an accessible extended region for a certain scope. Consider the following example:

```
{ region child @ parent;
  { region h = xopen(child);  // consume one lock capability
                  ...
  } // restore lock capability
  xdec(child);
}
```

The `xopen` construct *borrows* exactly one lock capability from the extended region ‘*child* for the scope of the `xopen` construct. The type system requires that region ‘*child* is *live* by the end of the `xopen` scope and creates a fresh logical region ‘*h*, which can be used as a traditional Cyclone region. It should be noted that ‘*child* is *still* live and possibly accessible (if it has more than one lock capability) during the scope of `xopen`. On the downside, region ‘*child must* remain locked for the scope of `xopen`.

4.10 Memory consistency

Our formal language semantics assumes a *sequentially consistent* memory model [25], which implies that concurrent read and write operations are viewed as an interleaving of *atomic* steps.

Modern, processors are implemented with much weaker memory consistency specifications as sequential consistency restricts common compiler and hardware optimizations. Research on re-

laxed memory models [1, 20] has shown that *race-free*[9] programs running on relaxed memory systems have a sequentially consistent view of memory operations.

Assuming that the compilation process preserves the original Cyclone code semantics, then we obtain *race-free* native code with sequential consistency guarantees. At the implementation level, we must guarantee that memory operations to extended regions cannot escape the scope of a "lock/unlock" primitive as locking operations *synchronize memory*. This situation may arise as a result of compiler optimizations such as *register promotion* [7]. We have taken the most conservative approach and require that extended region data objects are compiled down to C as *volatile*.[10] According to the GCC manual, the "*implementation is free to reorder and combine volatile accesses which occur between sequence points, but cannot do so for accesses across a sequence point*" [17]. Our locking primitives introduce sequence points and thus the compilation process will not reorder volatile accesses in an unsafe manner.

5. Implementation

5.1 Compiler

We have implemented extended region checking as a separate compiler pass in Cyclone. First, the type well-formedness of our annotations (effects, exceptions, types) is checked. During type checking, we disregard control-flow and verify that the extended regions being accessed exist in a function's scope (i.e., by inspecting the `@ieffect` annotation and tracking fresh regions). This allows us to catch common errors early. Once type checking is finished, the compiler enters the static analysis stage where it performs data- and control-flow analyses and determines candidate program locations for dynamic check (e.g., array bounds checks) insertion.

The compiler may eliminate some candidate locations, by utilizing programmer-inserted checks (e.g, `if` $(i < len)$ $a[i] = 10$;). As illustrated in Section 2, some optimizations may be unsound. Ideally, the data-flow analysis should discard programmer-inserted assertions for memory accesses at some region r, when they are followed by an unlock operation on r. Our current implementation, is highly conservative and only allows dynamic check elimination for trivial cases of shared memory accesses.

The exception analysis considers compiler-inserted checks as *implicit* exceptions and performs a control-flow sensitive analysis to verify that uncaught exceptions that *may* be thrown from a function body are included in the function's `@throws` specification.

Finally, a control-flow sensitive effect analysis is performed. The analysis propagates effects through the control flow graph and verifies that the output effect of the function body matches the function's `@oeffect` specification. It is worth noting that conditional, iterative and exception handling statements are treated *conservatively*. However, it is entirely possible to spawn a new thread, which consumes some locks or regions, in one of the branches of a conditional statement and not in the other. Of course, the programmer must insert a `cap` operation to the branch not containing the spawn operation, so as to guarantee that the output effects of both branches match. This analysis also utilizes function attributes, when checking function calls. For instance, effects are not propagated from function calls that never return to the calling context (i.e., `__attribute((noreturn))__`).

5.2 Code generation and run-time system

As discussed in Section 3.2, each thread maintains a local view of the hierarchy and each function call accesses a specific portion of the thread's view. This is a crucial feature for avoiding false region sharing, which in turn reduces parallelism, and memory leaks caused by bulk region deallocation.

Consider the case where a thread owning an unlocked region ρ *shares* that region with a new thread, which in turn allocates a fresh region ρ_1 at region ρ. The second thread uses ρ_1 locally and then deallocates it. Assuming that there is a global view of the hierarchy, ρ_1 is *owned* by the second thread. If the first thread attempts to lock ρ, then it will have to block until ρ_1 is deallocated by the second thread.

Deallocating regions *en masse* may temporarily cause a *region leak*, when there is an inexact correspondence between the dynamic hierarchy and the static hierarchy accessible to a function call. Let us assume that function calls access dynamically a single (local) view of the hierarchy, but each call views *statically* a specific *portion* of the hierarchy (including region and lock counts). At the type-level, when a non-leaf node of a hierarchy is removed from a function's effect, then the entire subtree of that node is also removed from the effect. The dynamic semantics can *only* decrement the count of the node being removed, but it cannot decrement the remaining nodes as it is unaware of the portion being removed. Thus, the subtree would temporarily leak until the non-leaf node is entirely removed. If the node being removed is pure, then it is safe to deallocate its subtree from the local hierarchy without requiring additional information. Otherwise, the compiler has two options: issue a warning about a possible leak or generate code that dynamically tracks the hierarchy passed to a function call. That is, the compiler could preserve an exact correspondence between run-time views and static effects. The advantage of the second approach is that it prevents temporary memory leaks. On the downside, it places an overhead for each function call that uses non-trivial hierarchies.

The current implementation strictly adheres to the formal semantics and implements the second option. As an optimization, we avoid code generation for function calls that use hierarchies of height one. However, it is entirely possible to allow the programmer to decide whether such leaks should be prevented, by adding annotations to functions (e.g., `@noleak`), or by introducing new compiler flags.

In the paragraphs that follow we discuss how the code generator assists the run-time system with type information so that it can prevent *false sharing* and *region leaks*. We also discuss about new features that have been added to the run-time system.

Code generation. We have altered the code generation pass so that we can perform the following tasks:

- Translate `spawn` statements to low-level primitives, which require (un)packing of function arguments and placing the call into a wrapper function, which acts as a glue between the call and thread that will executing it.
- Generate specialized code for allocating extended regions and references.
- Generate code for allowing "dynamic effect tracking" before *some* function calls. The sub-tree passed to a call is not actually copied. Instead we use a form of dynamic scoping (shallow binding in particular) so as to map type-level region names to nodes of the local tree. Each dynamic scope is pushed into the virtual stack frame of the run-time system. An additional `pop` statement is added after the call.

Run-time system. In order to maintain a local view of the global hierarchy, the run-time system performs the following tasks:

- It registers fresh regions to the local thread hierarchy in which they are allocated.

[9] Read and write operations to shared memory locations only occur within memory synchronization primitives.

[10] GCC is invoked by the Cyclone compiler to generate native code.

lang	CPU (s)	memory (KB)	load per core (%)				elapsed (s)	factor
Benchmark: **binary-trees**								
gcc	21.34	100,688	46	88	50	89	7.54	1.00
cyc	23.88	122,412	73	81	88	79	7.21	0.96
Benchmark: **chameneos-redux**								
gcc	56.38	576	68	90	72	91	17.04	1.00
cyc	276.08	1,112	85	99	84	100	76.69	4.50
Benchmark: **fannkuch**								
gcc	152.69	572	97	100	97	97	39.18	1.00
cyc	177.28	1,032	99	99	100	99	44.63	1.14
Benchmark: **mandelbrot**								
gcc	24.24	28,260	100	99	100	100	8.13	1.00
cyc	46.21	32,176	95	95	95	97	16.15	1.99
Benchmark: **thread-ring**								
gcc	143.09	4,536	26	38	24	16	133.04	1.00
cyc	254.38	18,108	13	49	22	16	246.95	1.86

Table 1. Performance overhead, compared to GCC, for five benchmarks taken from "The Computer Language Benchmarks Game."

- When a subtree has to be deallocated, it uses the dynamic scoping structures to retrieve nodes of the local hierarchy and update their dynamic counts accordingly. Notice that all remaining region locks are released during the deallocation phase.
- The implementation of `spawn` uses a similar technique to construct the subtree passed to new thread and makes this tree accessible to the new thread. It also performs capability accounting tasks so that the dynamic trees of both threads match the static effects.
- Region locking is implemented in a straightforward manner by traversing the local hierarchy. To avoid deadlocks, subtrees are always locked in a top-down left-to-right manner.
- The region allocation subsystem has been re-engineered so that it can serve concurrent allocation requests in a non-blocking manner (i.e., using atomic operations).

6. Performance evaluation

We evaluated our implementation on five concurrent benchmark programs, taken from "The Computer Language Benchmarks Game" (`http://shootout.alioth.debian.org/u32q/`). As a basis for our evaluation we used the fastest version of the programs in C, with one exception mentioned below, which we translated to our language as directly as possible. The results are summarized in Table 1. The five benchmark programs were:[11]

binary-trees a program that allocates, traverses and deallocates many binary trees. The original program (#7) uses GCC OpenMP and memory pools, as implemented in the Apache Portable Runtime Library.

chameneos-redux a program that simulates the interaction of a number of creatures, using symmetrical thread rendez-vous. Our basis for the comparison is the second fastest version in C (#2); it uses pthreads and mutex locks. On our testing machine, it only produced the correct result when compiled with `-O2` and we compiled our program with the same option. The fastest version in C (#5) uses the processor's "compare and swap" instruction, instead of locks, and explicitly schedules threads to processor cores; it cannot be translated directly to our language.

[11] Our implementation and the benchmark programs are available from: `http://www.softlab.ntua.gr/~pgerakios/cyc_reglock.tgz`.

fannkuch a program that performs indexed access to small sequences of integer numbers. The original program (#2) uses pthreads. On our testing machine, it only produced the correct result when compiled without optimizations and we did the same for our program.

mandelbrot a program that plots a bitmap of the Mandelbrot set. The original program (#6) uses pthreads and special SSE2 128-bit floating-point instructions. Our translation implements the same algorithm but is based on a simpler C# version of the program, using normal double precision numbers.

thread-ring a program that creates a large number of threads, organized in a ring, and repeatedly passes a token from one thread to the next. The original program (#1) uses pthreads and mutex locks. (We should mention that the original C program performs very poorly, compared to versions in other languages.)

The testing machine is a quad-core 2.4GHz Intel, with 2GB of RAM, running a Linux 2.6.30 kernel. Our implementation used GCC 4.3.2 as a back end, which was also used to compile the C programs. We used `-O3`, except as explained above. In our Cyclone implementation we disabled Boehm's garbage collector, which is only used for Cyclone's original regions and is not need for these benchmarks.

As shown in Table 1, the benchmark programs fall in three categories. First, in binary-trees and fannkuch, the Cyclone program runs a little faster (7%) and a little slower (15%) than the original C program, respectively. Especially for the case of binary-trees, this result is particularly interesting as the two compared programs use both the same algorithm and the same region-based memory management scheme. Second, in mandelbrot and thread-ring, the Cyclone program runs almost twice as slow, by a factor of 86% and 99%, respectively. In the case of mandelbrot, this is attributed to our Cyclone's inability to exploit the CPU's specialized number crunching instructions and, we believe, is not very interesting for the purposes of this paper. On the other hand, thread ring is an extreme case of benchmark, stressing thread communication; we believe that we can achieve better results here by further tuning the performance of our locks.

Third, we observe a very heavy performance penalty in chameneos-redux, which runs 4.5 times slower than the original program. This is the most interesting of our benchmarks, as it reports an inherent limitation of locking supported by the type system. The original program uses one lock for the meeting place, where the creatures meet. In addition to this lock, our program also uses a second lock for the entire array holding the creatures' data. In our program, the array must be locked because it is not possible to convince the type system that the creature who waits in the meeting room *will never* access its data, but instead this data will be updated by its peer and therefore no data race will occur. We believe that it is this second lock which causes the performance penalty.

7. Future work

The benchmarks of the previous section show that Cyclone extended with our language constructs provides static memory safety guarantees without significantly compromizing performance compared with optimized (and unsafe) C. Still, there are plenty of optimizations and improvements that could be done to our implementation. Here, we identify the most important ones according to our current benchmarking experiences.

Waiting for threads. A lexically-scoped Cilk-like [14] construct for allowing parent threads to wait for the children threads to terminate would be highly desirable:

```
join {
```

```
    for (int i = 0; i < size; i++)
        spawn worker(a[i]);
}
```

The compiler and run-time system could utilize *join* information so as to make better scheduling decisions.

Read only data. The multiple readers, single writer lock idiom has proven to be invaluable. Concurrent applications often use data structures as read-only for certain parts of a concurrent computation. Our type system can straightforwardly be extended to support this idiom, by introducing a new dimension for our capabilities:

$$\kappa' := n, n \mid \overline{n, n}$$
$$\kappa := \texttt{ro}(\kappa') \mid \texttt{rw}(\kappa')$$

A region capability can be read only (i.e., $ro(\kappa')$) or read/write (i.e., $rw(\kappa')$), which permit read only operations or read/write operations to a region respectively.

Finer grained locking. For certain applications it would be preferable to associate locks to individual references as opposed to regions. It is possible to extend our system to support finer-grained locking by blurring the distinction between regions and references. That is, a fresh region effect could be assigned to new lockable references. In turn, this could be implemented by introducing a new language construct or utilizing existing methods such as tracked pointers and existential types (i.e., a similar mechanism to dynamic regions). At run-time, the new reference would be allocated in the parent region's space and explicit deallocation would still be possible by using reaps [28].

Run-time system improvements. Undoubtedly, there are plenty of optimizations that could be performed in our run-time system. However, we strongly believe that the most important feature that should be added is scheduling support for efficiently mapping kernel threads to processors. A desirable feature would be to schedule tightly-coupled threads sharing the same regions on the same processor. The integration of cooperative threads (and possibly adding language support for such threads) would be highly desirable as the cooperative model seems to be highly scalable for event-based applications [30].

8. Related work

The first statically checked stack-based region system was developed by Tofte and Talpin [29]. Since then, several memory-safe systems that enabled early region deallocation for a sequential language were proposed [2, 13, 23, 31]. RC [15] and Cyclone [22] were the first imperative languages to allow safe region-based management with explicit constructs. 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 multi-threaded programs. Grossman proposed a type system for safe multi-threading in Cyclone [21]. Race freedom is guaranteed by statically tracking locksets within lexically-scoped synchronization constructs. Grossman's proposal allows for fine-grained locking, but does not enable early release of regions and locks and provides no support for data migration or lock transfers. In contrast, we provide such support and also support bulk region deallocation and hierarchical locking, as opposed to just primitive locking.

Statically checked region systems have also been proposed [9, 32, 33] for real-time Java to rule out dynamic checks imposed by its specification. Boyapati *et al.* [9] have introduced hierarchical regions in ownership types but the approach suffers from similar disadvantages as Grossman's work. Additionally, their type system only allows sub-regions for *shared* regions, whereas we do not have this limitation. In previous work, Boyapati *et al.* also proposed an ownership-based type system that prevents deadlocks and data races [8]; in contrast to that system, we support locking of arbitrary nodes in the region hierarchy. Static region hierarchies (depth-wise) have been used by Zhao *et al.* [33]. Their main advantage is that programs require fewer annotations compared to programs with explicit region constructs. In the same track, Zhao *et al.* [32] 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.

At the program verification side, Concurrent C minor [24] is a concurrent version of C with threads, shared memory and first-class locks, which uses a variant of separation logic to reason about programs. Even though, specifications in separation logic are finer-grained and more flexible than our type system, they require interactive proofs.

Many systems, such as Safe-C [6], CCured [26] and Deputy [11], aim to make C code safe. Some of these systems drop soundness guarantees, so as to reduce the annotation burden, require whole program analyses, which are not always possible in the presence of libraries and legacy code, or drop the explicit memory representation of C programs or provide no guarantees for concurrent programs. There have also been developed numerous static analyses for avoiding data races [27] and valid data sharing [4, 5] for C programs. Finally, there are numerous languages at the C level of abstraction with explicit concurrency features [3, 14, 16] but to the best of our knowledge none of them provides both memory safety and race freedom guarantees.

9. Concluding remarks

In this paper we presented the design and implementation of a formal language, employing region-based memory management and locking primitives, that provides memory safety and race freedom guarantees for well-typed concurrent programs. We discussed the integration of our formal language within Cyclone, argued about the decisions that we have made in order to guarantee memory safety and race freedom at the implementation level, and evaluated the performance of programs written in our language against highly optimized C programs. Although our implementation is still not very mature, the benchmark results that were reported are acceptable and promising.

The main contribution of our work is the development of an extension to Cyclone that supports race-free and memory-safe multi-threading. To the best of our knowledge, our language is the first variant of Cyclone that has been implemented with these properties, and one of the very few programming languages at this level of abstraction that has been designed and implemented with this goal in mind.

References

[1] S. V. Adve and M. D. Hill. Weak ordering – a new definition. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 2–14, New York, NY, USA, 1990. ACM.

[2] 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.

[3] T. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, and D. Zhang. Pillar: A parallel implementation language. In *Proceedings of the International*

Workshop on Languages and Compilers for Parallel Computing, pages 141–155, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Z. R. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–158, New York, NY, USA, 2008. ACM.

[5] Z. R. Anderson, D. Gay, and M. Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 98–109, New York, NY, USA, 2009. ACM.

[6] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, New York, NY, USA, 1994. ACM.

[7] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press.

[8] 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.

[9] 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.

[10] 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, 2003.

[11] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In R. De Nicola, editor, *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 4421 of *LNCS*, pages 520–535. Springer, 2007.

[12] 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.

[13] 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, 2006.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.

[15] 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, 2001. ACM Press.

[16] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[17] *GCC Online Documentation*. http://gcc.gnu.org/onlinedocs/.

[18] P. Gerakios, N. Papaspyrou, and K. Sagonas. A concurrent language with a uniform treatment of regions and locks. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, Electronic Proceedings in Theoretical Computer Science, 2009. To appear.

[19] P. Gerakios, N. Papaspyrou, and K. Sagonas. Race-free and memory-safe multithreading: Design and implementation in Cyclone. Technical report, National Technical University of Athens, 2009. http://www.softlab.ntua.gr/~pgerakios/papers/reglockimp_techrep10.pdf.

[20] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[21] 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, 2003. ACM Press.

[22] 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, 2002. ACM Press.

[23] 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.

[24] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.

[25] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Prog. Lang. Syst.*, 1(1):84–97, 1979.

[26] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Prog. Lang. Syst.*, 27(3):477–526, 2005.

[27] P. Pratikakis, J. S. Foster., and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, New York, NY, USA, 2006. ACM.

[28] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in Cyclone. *Sci. Comput. Program.*, 62(2):122–144, 2006.

[29] 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, 1994. ACM Press.

[30] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the Conference on Hot Topics in Operating Systems*, page 4, Berkeley, CA, USA, 2003. USENIX Association.

[31] 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.

[32] 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.

[33] 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.