ISBN 978-83-60810-22-4

Memory Safety and Race Freedom in Concurrent Programming Languages with Linear Capabilities

Niki Vazou
Email: nvazou@softlab.ntua.gr

Michalis Papakyriakou
Email: mpapakyr@softlab.ntua.gr

Nikolaos Papaspyrou
Email: nickie@softlab.ntua.gr

School of Electrical and Computer Engineering
National Technical University of Athens
Polytechnioupoli, 15780 Zografou, Athens, Greece

***Abstract*—In this paper we show how to statically detect memory violations and data races in a concurrent language, using a substructural type system based on linear capabilities. However, in contrast to many similar type-based approaches, our capabilities are not only linear, providing full access to a memory location but unshareable; they can also be read-only, thread-exclusive, and unrestricted, all providing restricted access to memory but extended shareability in the program source. Our language features two new operators, let! and lock, which convert between the various types of capabilities.**

I. Introduction

Multi-core computers have emerged as a new wave of technology and dictate the usage of concurrent programming languages. However, shared-memory concurrency further complicates existing problems, such as how to guarantee *memory safety*, and introduces new problems, such as how to avoid *data races*. Languages in the family of ML provide a good compromise between functional and imperative characteristics; their functional nature helps in restricting problems such as the above and their strong static type system helps detecting such problems at compile time, thus saving testing time and resources. However, when ML-style references are combined with concurrency, both problems remain and cannot be tackled by ML's type system.

As far as memory safety is concerned, most problems stem from aliasing. Many approaches have been proposed to control aliasing, the most direct of which are based on Girard's linear logic [7]. Linear types systems [22] have been used for region-based languages [21], [14], for a Lisp dialect [1], and for Cyclone [9], [20], a safe dialect of C, which uses a light-weight version of linearity in the form of tracked (unique) pointers. Our work is based on a linear language with locations [13], in which aliasing is controlled via linear capabilities.

A *capability* is a communicable, unforgeable token of authority which refers to a memory location and provides an associated set of access rights. In a linear language with locations and capabilities, operator new allocates a new memory location ℓ and returns, apart from a reference to the location ℓ, a linear capability for it. This capability must be provided for any further access to that memory location. However, capabilities are *linear* objects, which means that they can be used exactly once. Every operator that reads or writes to a memory location consumes the capability provided and, to enable further access to the location, must return a new one. Finally, operator free consumes the capability, rendering further access to that memory location impossible.

Data races are one of the most frequent sources of bugs in programs written in concurrent languages with shared memory. A data race occurs when two threads concurrently access the same data without synchronization, and at least one of the accesses is a write. Being time-dependent, data races can be one of the most difficult programming errors to detect, reproduce, and eliminate and this is why many researchers have implemented tools for their detection [16]. Such tools are either dynamic and lockset-based [17], [3] or static, type-based [2], [6], [5].

A direct way to detect data races in a concurrent language is to use a communication channel through which a thread can send a linear capability to another thread. If a thread owns a linear capability for a memory location ℓ and sends it to another thread via a channel, this is equivalent to the first thread unlocking ℓ and the second thread locking it. The π-calculus [12] supports communication via channels and can be extended to support linearity, thus forming a language that guarantees the absence of data races [11]. As an alternative, virtual communication channels can be constructed from linear operators [18].

Using channels directly to exchange capabilities is a burden for programmers, so other more indirect approaches have been proposed. Shi and Xi [19] have proposed a type system based on a notion of types with effects, where a special modality supports the sharing of linear resources. Ennals *et al.* [4] have proposed an imperative concurrent language for packet processing applications. They use a simple linear type system to ensure that no packet (representing a memory region) can be referenced simultaneously by multiple threads. This constraint is arguably not too restrictive, as statistically "processing of a packet by multiple threads simultaneously is rare." When allocating a packet, a thread acquires a linear handler for it. Each child of this thread can access the packet with operations that either need a linear handler (i.e., free or strong updates) or an unrestricted one (i.e., dereference or weak updates). If a child needs a linear handler, it gains it and that packet is no longer visible by the parent thread. Otherwise, the child gains an unrestricted copy of the handler, which is alive inside its body. At runtime, this is equivalent to implicitly locking the packet. Unlocking will be performed by thread's termination.

A less restrictive approach has been proposed by Wittie and Lockhart [24]. They start with the core language λ^{low} [10], a sequential language whose linear type system guarantees memory safety, based on Walker's and Watkins language with linear regions [23]. On top of this, they build $\lambda^{concurrent}$ which provides concurrency and a locking mechanism, based on capabilities. As linear capabilities can be used by exactly one thread, they introduce the notion of *lock*, which can be thought of as a non-linear capability that can be shared between several threads. A lock can be *created* by consuming a linear capability for a memory location. Later, a lock can be *acquired*, producing again the linear capability from which it was created; however, the runtime semantics ensures that at most one thread has acquired the lock at any time. When a thread that acquired the lock does not need it anymore, it can *release* it, thus consuming again the linear capability and enabling other threads to acquire the lock. A disadvantage of this mechanism is that once a lock is created for a memory location, it is not possible any more to safely deallocate this memory location.

A different approach, using a type and effect system to guarantee data race freedom, has been proposed by Gerakios *et al.* [6]. Their work extends Cyclone [8], a safe dialect of C, with concurrency. They use capabilities that are annotated with two counts, the region and the lock count, which denote whether a region is live and locked respectively. Incrementing a lock count from 0 to 1 amounts to acquiring a region lock, while decrementing counts amounts to releasing it; numbers bigger than one can be used to support aliasing and re-entrant locks. Moreover, their system includes *impure* capabilities $(\overline{n_1, n_2})$, which are obtained by splitting pure or other impure capabilities in pieces, in order to pass them to multiple threads (e.g., the pure capability $(3, 2)$ can be split into two impure capabilities $(\overline{2, 1})$ and $(\overline{1, 1})$). An impure capability denotes that a thread's knowledge of a region's counts is inexact. The collaboration of the two kinds of capabilities with the runtime system ensures that regions are safely deallocated. A disadvantage of this system, however, is that it does not support read/write locks.

In this work we propose a type system with annotated capabilities for detecting memory violations and data races. Our language is based on L^3 [13] extended with the let! operator [15] and a lock operator. These operators are used to annotate capabilities not only as linear or unrestricted, but also, as *read-only* or *thread-exclusive*. The typing rules ensure that each thread-exclusive value can be visible by exactly one thread. One of the main advantages of our system is that it permits concurrent read-only access to the same memory location, without reporting false positive data races. In Section II we describe our language through examples, whereas in Section III we formally define the language's syntax, typing, and operational semantics. We finish with some concluding remarks.

II. An informal description

We use linear capabilities to guarantee memory safety and race freedom in a shared memory concurrent language. To this end, we use an appropriate set of qualifiers that provide specific privileges for memory access. Qualifiers are used in languages where linear and unrestricted values coexist, in order to distinguish between these two. In our language, only capabilities are meant to be qualified. However, as capabilities can be stored in pairs or existential packages, and used by function closures, all these kinds of values must also be qualified, in order to prevent the abuse of capabilities.

In languages with capabilities that can only be linear, there are two big disadvantages. Once a linear capability is used, it is automatically consumed; therefore, all operators that do not mean to consume a capability (e.g., read and write) must return a new instance of the capability, and programming becomes awkward. Moreover, linear capabilities cannot be duplicated; therefore they cannot be shared between threads (or other independent parts of the program) that are meant to have the same access privileges. To overcome both disadvantages, several languages with linear capabilities provide mechanisms for controllably transforming linear objects to unrestricted ones, such as Wadler's let! [21], Odersky's observer types [14], or the freeze and thaw operators in L^3 [13].

For the same purpose, in our language we use the scope-based operator let! $(x = e)$ as s at ρ then $y = e_1$ in e_2, which has been presented in our previous work [15] in a setting with just two states: L and U. This operator evaluates e and binds its linear value to the variable x. This variable is used during the evaluation of e_1 with a non-linear qualifier; it is then reinstated as linear during the evaluation of e_2. Also, the result of e_1 is bound to the variable y, which may be used in e_2. In order to control the uses of the non-linear x and avoid, for instance, that it escapes in the result of e_1 or through a function closure, the let! operator introduces a new (type-level) scope ρ, which is valid only in the context of e_1, and annotates the type of the non-linear version of x with this scope.

To be more specific, in our language a qualifier q is of the form $s \,\mathsf{at}\, \pi$, where s is a state and π is the qualifier's scope (which can be a type-level variable ρ or the special scope $\perp$). There are four possible states:

- *Unrestricted* (U): An unrestricted capability for a memory location provides no privileges for that location. It can be freely shared.
- *Read-Only* (R): A read-only capability for a memory location provides only read access for that location. It can be freely shared.
- *Thread-Exclusive* (T): A thread-exclusive capability for a memory location provides write and read access for that location. It can be shared in the context of a single thread.
- *Linear* (L): A linear capability for a memory location provides access to deallocate that location. It can never be shared.

A partial order $\sqsubseteq$ is defined on states by $\mathsf{L} \sqsubseteq \mathsf{T} \sqsubseteq \mathsf{R} \sqsubseteq \mathsf{U}$. In the operator let! $(x = e)$ as s at ρ then $y = e_1$ in e_2, the non-linear version of x is qualified by $s\,\mathsf{at}\,\rho$, for some $s \neq \mathsf{L}$.

When a new location is created, its capability starts with a qualifier $\mathsf{L}\,\mathsf{at}\,\bot$. (From now on, we will use s as an abbreviation for $s\,\mathsf{at}\,\bot$, to simplify presentation.) However, before this capability can be used for any other purpose than deallocating the location, it must be converted to a non-linear qualifier using the let! operator. In this way, its qualifier is "downgraded" but the capability can now be shared. For instance, if the new qualifier has a state of R or U, the capability can be shared among various concurrent threads. Notice however that if one of these threads needs to have write access to the location, a qualifier of state T will be required. The symmetric operator lock $(x = e)$ as s at ρ then $y = e_1$ in e_2 performs this qualifier "upgrading." It evaluates e and binds its non-linear value to the variable x. This variable is used during the evaluation of e_1 with a qualifier of $s\,\mathsf{at}\,\rho$ (where $s \neq \mathsf{L}$); it is then reinstated to its previous qualifier during the evaluation of e_2. Also, the result of e_1 is bound to y for use in e_2. In contrast to the let! operator, the lock! operator must make sure (at runtime) that no other thread possesses a conflicting capability, e.g., if a thread-exclusive lock for some location ℓ is requested, that no other thread possesses a T or R capability for ℓ.

We present our language informally with a series of examples. In the rest of this section, we liberally extend the formal language that will be presented in Section III with features that are orthogonal to what we present there and could easily be introduced. Most notably, we use integer and boolean values (types Int and Bool respectively) and assorted operators; we also use recursive functions, defined with a letrec construct.

In the simplest example, a thread allocates a memory location, accesses its contents and deallocates it.

Example 1

```
let $\ulcorner \ell, p \urcorner$ = new 0 in
let $(c, r) = p$ in                        // $c : {}^{\mathsf{L}}\mathsf{Cap}\ \ell\ \mathsf{Int}$, $r : \mathsf{Loc}\ \ell$
let! $(x = c)$ as T at $\rho$ then           // $x : {}^{\mathsf{T\,at}\,\rho}\mathsf{Cap}\ \ell\ \mathsf{Int}$
  $y =$
    $(x, r) := !(x, r) + 1$
in
  free $\ulcorner \ell, (x, r) \urcorner$                  // $x : {}^{\mathsf{L}}\mathsf{Cap}\ \ell\ \mathsf{Int}$
```

The expression new 0 allocates a new memory location and returns an existential package of type ${}^{\mathsf{L}}\mathsf{Xref\ Int}$, where

$${}^{q}\mathsf{Xref}\ \tau \;\equiv\; {}^{q}\exists\ell.\,{}^{q}\mathsf{Lref}\ \ell\ \tau$$

We immediately open the package, whose contents are a type-level variable ℓ (a type-level abstraction of the new memory location) and a pair p of type ${}^{\mathsf{L}}\mathsf{Lref}\ \ell\ \mathsf{Int}$, where

$${}^{q}\mathsf{Lref}\ r\ \tau \;\equiv\; {}^{q}\langle {}^{q}\mathsf{Cap}\ r\ \tau * \mathsf{Loc}\ r\rangle$$

The contents of this pair are a pointer r to the new memory location and a linear capability c for this. Notice that, to unpack the pair p we use the construct let $(x, y) = e_1$ in e_2 which extracts both components by consuming the pair once, as required in languages that support linearity.[1]

[1]To make presentation simpler, in the examples of this section we often omit the external qualifier when constructing pairs, as it can easily be deduced from the pair's contents. In the strict syntax of Section III, the assignment in the first example would be ${}^{\mathsf{T}}(x, r) := !^{\mathsf{T}}(x, r) + 1$.

The let! operator is then used to downgrade the capability c to thread-exclusive (T) and store the downgraded capability in x. This downgraded capability provides write and read access and can be used multiple times, in the first clause of let!. Both the capability x and the location r are provided to enable access to that location, in both uses of operators := (assignment) and ! (dereference); variable y, the result of the assignment, is not used. Finally, in the second clause of let! the capability x is reinstated to linear and the free operator is used to consume it. Operator free is the complement of new, taking an argument of type ${}^{\mathsf{L}}\mathsf{Xref}\ \tau$, deallocating the memory location and returning the contents that were stored in it.

In a second example, we downgrade the capability from linear to read-only (R), utilizing the let! construct, in order to share it among a couple of new threads we create.

Example 2

```
                                           // $c : {}^{\mathsf{L}}\mathsf{Cap}\ \ell\ \mathsf{Int}$, $r : \mathsf{Loc}\ \ell$
let! $(x = c)$ as R at $\rho$ then           // $x : {}^{\mathsf{R\,at}\,\rho}\mathsf{Cap}\ \ell\ \mathsf{Int}$
  $y =$
    $\ldots\ !(x, r)\ \ldots\ \|\ \ldots\ !(x, r)\ \ldots$
in
  $\ldots$
```

We should note that if the downgrade was to thread-exclusive (T), this program would not typecheck because capabilities of state T cannot be shared among multiple threads.

Nonetheless, a thread is able to upgrade a capability's state and gain read or write access to a memory location, as the following example shows.

Example 3

```
                                     // $c_1 : {}^{\mathsf{R\,at}\,\rho_1}\mathsf{Cap}\ \ell_1\ \tau$, $r_1 : \mathsf{loc}\ \ell_1$
                                     // $c_2 : {}^{\mathsf{R\,at}\,\rho_2}\mathsf{Cap}\ \ell_2\ \tau$, $r_2 : \mathsf{loc}\ \ell_2$
lock $(x = c_1)$ as T at $\rho$ then          // $x : {}^{\mathsf{T\,at}\,\rho}\mathsf{Cap}\ \ell_1\ \tau$
  $y =$
    $(x, r_1) := !(c_2, r_2)$
in
  $\ldots$
```

In this example, a thread is given two read-only capabilities for locations ℓ_1 and ℓ_2. Before it can update the contents of ℓ_1, it has to acquire a thread-exclusive capability for it. As we already mentioned, at runtime the lock operator ensures that no other thread possess a R capability for ℓ_1 before proceeding.

As a last and more involved example, we describe how a synchronous producer-consumer program can be formalized in our language. As an abbreviation, again, to simplify presentation we omit the qualifier U from the types of functions, monomorphic or polymorphic. We also assume the existence of the following functions:

produce	:	$\mathsf{Unit} \to \mathsf{Int}$
consume	:	$\mathsf{Int} \to \mathsf{Unit}$
empty	:	$\mathsf{Unit} \to \mathsf{Bool}$
write	:	$\forall\rho.\ \mathsf{Int} \to {}^{\mathsf{U\,at}\,\rho}\mathsf{Cap}\ \ell\ \mathsf{Int} \to \mathsf{Loc}\ \ell \xrightarrow{[\rho]} \mathsf{Unit}$
read	:	$\forall\rho.\ {}^{\mathsf{U\,at}\,\rho}\mathsf{Cap}\ \ell\ \mathsf{Int} \to \mathsf{Loc}\ \ell \xrightarrow{[\rho]} \mathsf{Int}$

Functions *produce* and *consume* are abstractions for the actual producing and consuming of integer values; *empty* returns true if a produced value is waiting to be consumed; *write* and *read* are primitives for storing and retrieving produced values, keeping track of the empty state.

The two recursive procedures *producer* and *consumer* form the main core of the program.

Example 4

```
                                        // $c$ : ${}^{\mathsf{U\,at}\,\rho}\mathsf{Cap}\ \ell\ \mathsf{Int}$, $r$ : $\mathsf{Loc}\ \ell$
letrec $producer = {}^{\mathsf{U}}\lambda d : \mathsf{Unit}.$
  if $empty$ unit then
    let $x = produce$ unit in
    $write\,[\rho]\; x\; c\; r$;
    $producer$ unit
  else
    $producer$ unit
in
letrec $consumer = {}^{\mathsf{U}}\lambda d : \mathsf{Unit}.$
  if not ($empty$ unit) then
    let $x = read\,[\rho]\; c\; r$ in
    $consume\ x$;
    $consumer$ unit
  else
    $consumer$ unit
in
$producer$ unit $\parallel$ $consumer$ unit
```

The producer function tests whether the memory location ℓ is empty; if it is, it produces a value x, writes it to the memory location ℓ and recursively calls itself, otherwise it busy-waits. Similarly, the consumer function tests if a produced value is waiting in location ℓ; if it is, it reads x from the memory location ℓ and consumes it, otherwise it busy-waits.

The producer function attempts to gain a thread-exclusive lock before actually writing to ℓ and, similarly, the consumer function attempts to gain a read-only lock before actually reading from the memory location. These two are implemented by functions *write* and *read*, which can be defined as follows. We assume the existence of two functions *setNonEmpty* and *setEmpty* which cooperate with function *empty*.

```
                              // $setNonEmpty$ : Unit $\to$ Unit
                                 // $setEmpty$ : Unit $\to$ Unit
$write$ = ${}^{\mathsf{U}}\Lambda\rho.\,{}^{\mathsf{U}}\lambda v : \mathsf{Int}.\,{}^{\mathsf{U}}\lambda c : {}^{\mathsf{U\,at}\,\rho}\mathsf{Cap}\ \ell\ \mathsf{Int}.\,{}^{\mathsf{U}}\lambda r : \mathsf{Loc}\ \ell.$
          lock $(x = c)$ as T at $\rho'$ then
            $y =$
              $(x, r) := v$
          in
            $setNonEmpty$ unit
$read$  = ${}^{\mathsf{U}}\Lambda\rho.\,{}^{\mathsf{U}}\lambda c : {}^{\mathsf{U\,at}\,\rho}\mathsf{Cap}\ \ell\ \mathsf{Int}.\,{}^{\mathsf{U}}\lambda r : \mathsf{Loc}\ \ell.$
          lock $(x = c)$ as R at $\rho'$ then
            $y =$
              $!(x, r)$
          in
            $setEmpty$ unit;
            $y$
```

$$
\begin{array}{lcl}
s & ::= & \mathsf{L} \mid \mathsf{T} \mid \mathsf{R} \mid \mathsf{U} \\
\pi & ::= & \rho \mid \bot \\
q & ::= & s\,\mathsf{at}\,\pi \\
r & ::= & \ell \mid i \\
\phi & ::= & \mathsf{Cap}\ r\ \tau \mid \langle \tau_1 * \tau_2 \rangle \mid \tau_1 \xrightarrow{\vec{\pi}} \tau_2 \mid \forall \rho.\,\tau \mid \exists \ell.\,\tau \\
\tau & ::= & \mathsf{Unit} \mid \mathsf{Loc}\ r \mid {}^{q}\phi \\
e & ::= & \mathsf{unit} \mid x \mid {}^{q}\lambda x : \tau.\,e \mid e_1\ e_2 \mid {}^{q}\Lambda\rho.\,e \mid e\ [\pi] \\
 & \mid & {}^{q}(e_1, e_2) \mid \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 \\
 & \mid & {}^{q}\ulcorner r, e \urcorner \mid \mathsf{let}\ \ulcorner l, x \urcorner = e_1\ \mathsf{in}\ e_2 \\
 & \mid & \mathsf{new}\ e \mid \mathsf{free}\ e \mid !e \mid e_1 := e_2 \\
 & \mid & e_1 ; e_2 \mid e_1 \parallel e_2 \\
 & \mid & \mathsf{let!}\ (x = e)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y = e_1\ \mathsf{in}\ e_2 \\
 & \mid & \mathsf{lock}\ (x = e)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y = e_1\ \mathsf{in}\ e_2 \\
 & \mid & \mathsf{loc}\ i \mid {}^{q}\mathsf{cap}\ i \\
 & \mid & \mathsf{let\$}\ (x = e)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y = e_1\ \mathsf{in}\ e_2 \\
 & \mid & \mathsf{lock\$}\ (x = e)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y = e_1\ \mathsf{in}\ e_2 \\
v & ::= & \mathsf{unit} \mid \mathsf{loc}\ i \mid {}^{q}u \\
u & ::= & \mathsf{cap}\ i \mid \lambda x : \tau.e \mid \Lambda\rho.\,v \mid (v_1, v_2) \mid \ulcorner i, v \urcorner
\end{array}
$$

Fig. 1. Syntax.

III. Formalism

A. The syntax

The language we use is a typed lambda calculus with ML-style references. It is polymorphic with respect to scope variables. We have kept the language as simple as possible, including only the features that are necessary to demonstrate our approach, in the light of the issues that we discussed in the previous sections.

The syntax of our language is shown in Fig. 1. Our expressions include unit, term variables, term and scope abstractions, application for terms and scopes, pairs, sequential and concurrent execution of two expressions. There are primitives to create a reference, to read or update its contents and to deallocate it. We also include existential packages, and a primitive to open them.

There are two constructs that manipulate capability qualifiers: the let! primitive, which downgrades a linear capability, and the lock primitive, which blocks thread execution until it is safe to grant the requested capability. The constructs let$ and lock$ are not available to the programmer, but appear only during the evaluation of a program. The same is true about the expressions for locations (loc i) and capabilities (cap i).

Types (τ) may be the unit type, location types or pretypes (ϕ) annotated by a qualifier. A qualifier consists of a state s and a scope π. The state can be L, T, R and U, for linear, thread-exclusive, read-only or unrestricted values, respectively. Scopes are either ρ for a scope variable or $\bot$, which is the external scope and is always valid. As mentioned earlier, not only capabilities must be annotated with qualifiers, but also every expression type that may contain a capability. Thus, besides capabilities, pretypes include pairs, functions, scope

$$\boxed{\Gamma; \Delta; Z; M \vdash e : \tau}$$

$$\frac{\textsf{state}\ \Gamma \neq \textsf{L}}{\Gamma; \Delta; Z; M \vdash \textsf{unit} : \textsf{Unit}} \qquad \frac{\textsf{state}\ \Gamma \neq \textsf{L} \quad \textsf{frv}\ (\tau) \subseteq \Delta \quad Z \models \textsf{scope}\ \tau}{\Gamma, x : \tau; \Delta; Z; M \vdash x : \tau}$$

$$\frac{s \sqsubseteq \textsf{state}\ \Gamma \qquad Z \models \rho \qquad \Gamma, x : \tau; \Delta; Z_e; M \vdash e : \tau_1}{\Gamma; \Delta; Z; M \vdash {}^{s\,\textsf{at}\,\rho}\lambda x : \tau.\, e : {}^{s\,\textsf{at}\,\rho}(\tau \xrightarrow{Z_e} \tau_1)} \qquad \frac{\Gamma_1; \Delta; Z_1; M \vdash e_1 : {}^{q}(\tau_1 \xrightarrow{Z_e} \tau_2) \qquad \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau_1 \quad Z \models \textsf{scope}\ \tau_2}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2 \cup Z \cup Z_e; M \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\text{fresh}\ \rho' \quad Z' = Z \vee Z, \rho' \quad Z_1 \models \rho_1 \qquad \Gamma; \Delta; Z'; M \vdash e[\rho \mapsto \rho'] : \tau \quad s \sqsubseteq \textsf{state}\ \tau}{\Gamma; \Delta; Z \cup Z_1; M \vdash {}^{s\,\textsf{at}\,\rho_1}\Lambda \rho.\, e : {}^{s\,\textsf{at}\,\rho_1}\forall \rho'.\, \tau} \qquad \frac{Z_\tau \models \textsf{scope}\ \tau \quad Z_\pi \models \pi \qquad \Gamma; \Delta; Z; M \vdash e : {}^{q}\forall \rho.\, \tau}{\Gamma; \Delta; Z \cup Z_\pi \cup Z_\tau; M \vdash e\,[\pi] : \tau}$$

$$\frac{\Gamma; \Delta; Z; M \vdash e : \tau}{\Gamma; \Delta; Z; M \vdash \textsf{new}\ e : {}^{L}\textsf{Xref}\ \tau} \qquad \frac{Z_\tau \models \textsf{scope}\ \tau \qquad \Gamma; \Delta; Z; M \vdash e : {}^{L}\textsf{Xref}\ \tau}{\Gamma; \Delta; Z \cup Z_\tau; M \vdash \textsf{free}\ e : \tau}$$

$$\frac{\Gamma_1; \Delta; Z_1; M \vdash e_1 : {}^{\textsf{T}\,\textsf{at}\,\pi}\textsf{Lref}\ r\ \tau \qquad \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau \quad \textsf{state}\ \tau \neq \textsf{L}}{\Gamma_1 \oplus \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash e_1 := e_2 : \textsf{Unit}} \qquad \frac{\Gamma; \Delta; Z; M \vdash e : {}^{s\,\textsf{at}\,\pi}\textsf{Lref}\ r\ \tau \qquad s = \textsf{R} \vee \textsf{T} \quad Z_\tau \models \textsf{scope}\ \tau}{\Gamma; \Delta; Z \cup Z_\tau; M \vdash\ !e : \tau}$$

$$\frac{\Gamma_1; \Delta; Z_1; M \vdash e_1 : \tau_1 \qquad \Gamma_2; \Delta; Z_2; M \vdash e_2 : \tau_2 \quad s = \textsf{LUB}(\textsf{state}\ \tau_1, \textsf{state}\ \tau_2)}{\Gamma_1 \odot \Gamma_2; \Delta; Z_1 \cup Z_2; M \vdash e_1 \parallel e_2 : {}^{s}\langle \tau_1 * \tau_2 \rangle}$$

$$\frac{\text{fresh}\ \rho' \quad \textsf{L} \sqsubset s \quad \Gamma; \Delta; Z; M \vdash e : {}^{\textsf{L}}\textsf{Cap}\ r\ \tau \quad Z_1' = Z_1 \vee Z_1, \rho' \qquad \Gamma_1, x : {}^{s\,\textsf{at}\,\rho'}\textsf{Cap}\ r\ \tau; \Delta; Z_1'; M \vdash e_1[\rho \mapsto \rho'] : \tau_1 \quad \Gamma_2, x : {}^{L}\textsf{Cap}\ r\ \tau, y : \tau_1; \Delta; Z_2; M \vdash e_2 : \tau_2}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; Z \cup Z_2 \cup Z_3; M \vdash \textsf{let!}\ (x = e)\ \textsf{as}\ s\ \textsf{at}\ \rho\ \textsf{then}\ y = e_1\ \textsf{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma; \Delta; Z; M \vdash e : {}^{s'\,\textsf{at}\,\pi}\textsf{Cap}\ r\ \tau \quad \textsf{L} \sqsubset s \sqsubseteq s' \quad \text{fresh}\ \rho' \quad Z_1' = Z_1 \vee Z_1, \rho' \qquad \Gamma_1, x : {}^{s\,\textsf{at}\,\rho'}\textsf{Cap}\ r\ \tau; \Delta; Z_1'; M \vdash e_1[\rho \mapsto \rho'] : \tau_1 \quad \Gamma_2, y : \tau_1; \Delta; Z_2; M \vdash e_2 : \tau_2}{\Gamma \oplus \Gamma_1 \oplus \Gamma_2; \Delta; Z \cup Z_1 \cup Z_2; M \vdash \textsf{lock}\ (x = e)\ \textsf{as}\ s\ \textsf{at}\ \rho\ \textsf{then}\ y = e_1\ \textsf{in}\ e_2 : \tau}$$

Fig. 2. Typing rules.

abstractions and existential packages. Function types are also annotated with a set of scopes that are used by the function body, much like in our previous work [15].

B. Typechecking

The typing relation for our language is $\Gamma; \Delta; Z; M \vdash e : \tau$. Some selected typing rules are presented in Fig. 2. Γ is the environment that binds variables to types, Δ is the set of live location variables, M binds locations to types (and is only needed for the metatheory), and Z is the set of live scopes.

To simplify the rules we have used the following abbreviations, which we already mentioned in Section II.

$$\begin{array}{lcl} {}^{q}\textsf{Xref}\ \tau & \equiv & {}^{q}\exists \ell.\, {}^{q}\textsf{Lref}\ \ell\ \tau \\ {}^{q}\textsf{Lref}\ r\ \tau & \equiv & {}^{q}\langle {}^{q}\textsf{Cap}\ r\ \tau * \textsf{Loc}\ r \rangle \end{array}$$

Typing judgments of linear languages differ from those of regular, unrestricted languages mainly in the way they handle typing environments. In our language, only values may be linear, hence special treatment must be made only for environment Γ. In the typing of a composite expression, this environment is split into an appropriate number of pieces and it is ensured that each linear variable appears in exactly one piece.

To this means, we define a union operator $\Gamma_1 \oplus \Gamma_2$, for term environments, which is valid only if the intersection of Γ_1 and Γ_2 does not contain any linear values. This operator is defined in Fig. 3. To prevent linear values from being discarded, the typing of all base cases restricts Γ to contain only the linear bindings that are actually used. In a similar way we ensure that each thread-exclusive value can appear exactly in one thread. To this means, we define one more environment operator, $\Gamma_1 \odot \Gamma_2$ (Fig. 3) which enforces that the intersection of Γ_1 and Γ_2 contains neither linear nor thread-exclusive values. According to the definition of this operator, read-only values can either be duplicated to both Γ_1 and Γ_2 or passed exclusively to one of them. The rationale behind this behaviour is to give us the flexibility to pass a read-only lock exclusively to one of two parallel expressions, in case the other one does not require it.

The state τ operator is defined in Fig. 4 and returns the state part of the qualifier of a type τ. We use function $\textsf{LUB}$ to denote the least upper bound according to the ordering defined by $\sqsubseteq$. This operator is used in the typing rules to ensure that a pair can never exploit a linear value. It is extended to state Γ, which returns the $\textsf{LUB}$ of all the types stored in the environment Γ. The scope τ returns the scope part of the

$$\boxed{\Gamma = \Gamma_1 \oplus \Gamma_2}$$

$$\frac{}{\emptyset = \emptyset \oplus \emptyset}$$

$$\frac{\Gamma = \Gamma_1 \oplus \Gamma_2 \quad s = \mathsf{T} \vee \mathsf{R} \vee \mathsf{U}}{\Gamma, x :^{s \,\mathsf{at}\, \pi} \phi = \Gamma_1, x :^{s \,\mathsf{at}\, \pi} \phi \oplus \Gamma_2, x :^{s \,\mathsf{at}\, \pi} \phi}$$

$$\frac{\Gamma = \Gamma_1 \oplus \Gamma_2}{\Gamma, x :^{\mathsf{L} \,\mathsf{at}\, \pi} \phi = \Gamma_1, x :^{\mathsf{L} \,\mathsf{at}\, \pi} \phi \oplus \Gamma_2}$$

$$\frac{\Gamma = \Gamma_1 \oplus \Gamma_2}{\Gamma, x :^{\mathsf{L} \,\mathsf{at}\, \pi} \phi = \Gamma_1 \oplus \Gamma_2, x :^{\mathsf{L} \,\mathsf{at}\, \pi} \phi}$$

$$\boxed{\Gamma = \Gamma_1 \odot \Gamma_2}$$

$$\frac{}{\emptyset = \emptyset \odot \emptyset}$$

$$\frac{\Gamma = \Gamma_1 \odot \Gamma_2 \quad s = \mathsf{R} \vee \mathsf{U}}{\Gamma, x :^{s \,\mathsf{at}\, \pi} \phi = \Gamma_1, x :^{s \,\mathsf{at}\, \pi} \phi \odot \Gamma_2, x :^{s \,\mathsf{at}\, \pi} \phi}$$

$$\frac{\Gamma = \Gamma_1 \odot \Gamma_2 \quad s = \mathsf{L} \vee \mathsf{T} \vee \mathsf{R}}{\Gamma, x :^{s \,\mathsf{at}\, \pi} \phi = \Gamma_1, x :^{s \,\mathsf{at}\, \pi} \phi \odot \Gamma_2}$$

$$\frac{\Gamma = \Gamma_1 \odot \Gamma_2 \quad s = \mathsf{L} \vee \mathsf{T} \vee \mathsf{R}}{\Gamma, x :^{s \,\mathsf{at}\, \pi} \phi = \Gamma_1 \odot \Gamma_2, x :^{s \,\mathsf{at}\, \pi} \phi}$$

Fig. 3. Sequential and concurrent union operator.

$$\boxed{\mathsf{state}\ \tau}$$

$$\mathsf{state}\ \mathsf{Unit} = \mathsf{U} \quad \mathsf{state}\ (\mathsf{Loc}\ i) = \mathsf{U} \quad \mathsf{state}\ ({}^{s \,\mathsf{at}\, \rho}\phi) = s$$

$$\boxed{\mathsf{state}\ \Gamma}$$

$$\frac{}{\mathsf{state}\ \emptyset = \mathsf{U}} \qquad \frac{\mathsf{state}\ \Gamma = s_1 \quad \mathsf{state}\ \tau = s_2}{\mathsf{state}\ (\Gamma, x : \tau) = \mathsf{LUB}(s_1, s_2)}$$

$$\boxed{\mathsf{scope}\ \tau}$$

$$\mathsf{scope}\ \mathsf{Unit} = \bot \quad \mathsf{scope}\ (\mathsf{Loc}\ i) = \bot \quad \mathsf{scope}\ ({}^{s \,\mathsf{at}\, \rho}\phi) = \rho$$

$$\boxed{Z \models \pi}$$

$$\emptyset \models \bot \qquad\qquad \{\rho\} \models \rho$$

Fig. 4. Auxiliary definitions in typechecking.

qualifier of a type τ and it is used to check that this scope is valid with respect to Z. We use the frv (τ) to gain all the free location variables that appear in τ. We use this operator to check that all free location variables of a term variable are valid according to Δ.

In the typing rules exposed in Fig. 2 one may notice that all reference related expressions require along with the reference the corresponding capability. Typing rules for term abstraction and application ensure that scopes used in the function closure will be alive during any application of this function. For that matter, we handle environment Z in a relevant way, thus avoiding scopes from being added if they are not actually used. Relevant treatment of Z is enforced by the *minimal scope* relation $Z \models \pi$ between scope π and scope environment

$$\boxed{S; x \Downarrow S'; v}$$

$$\frac{\mathsf{state}\ v = \mathsf{L}}{S, x \mapsto v; x \Downarrow S; v} \qquad \frac{\mathsf{state}\ v \neq \mathsf{L}}{S, x \mapsto v; x \Downarrow S, x \mapsto v; v}$$

$$\boxed{\mathsf{state}\ v}$$

$$\mathsf{state}\ \mathsf{unit} = \mathsf{U} \quad \mathsf{state}\ (\mathsf{loc}\ i) = \mathsf{U} \quad \mathsf{state}\ ({}^{s \,\mathsf{at}\, \rho}u) = s$$

Fig. 5. Auxiliary definitions in operational semantics.

Z, also defined in 4. The $\oplus$ split operator is used in every typing rule which contains subexpressions except the rule for parallel execution in which case the $\odot$ split operator is used. Typing rules for let! and lock expressions are pretty-much the same, varying only on the restriction between the current and requested qualifier scope.

C. Operational Semantics

We define a small-step, call-by-value operational semantics for our language.

Our evaluation rules require two kinds of annotations. Every expression must be annotated with its thread identifier and in every parallel expression, each of the subexpressions must be annotated with the set of locks it inherits from its parent thread. Thread identifiers are introduced during the evaluation, whereas lock set annotations can be inferred statically with the aid of our type system. To this end, we define a function $\mathsf{cl}\ e$ on expressions which, given the typing derivation, calculates a lock set for every parallel subexpression in e. It is defined recursively, treating the expression $e_1 \parallel e_2$ in the following way: $\mathsf{cl}\ (e_1 \parallel e_2) = {}^{cl_1}(\mathsf{cl}\ e_1) \parallel {}^{cl_2}(\mathsf{cl}\ e_2)$, where ${}^{s \,\mathsf{at}\, \rho}\mathsf{cap}\ l \in cl_i$ iff $(s = \mathsf{R} \vee \mathsf{T}) \wedge (\exists \tau)[{}^{s \,\mathsf{at}\, \rho}\mathsf{Cap}\ l\ \tau \in \Gamma_i]$, where Γ_i is the environment in which e_i was typechecked. By using our type system, we ensure that locks will be distributed among child threads in a safe manner. Here, by safe, we mean that *all* locks will be distributed and that thread-exclusive locks will be given to only one child thread.

Our semantics is a relation between configurations consisting of a store S, which is a mapping from variables to values, a memory μ which is a mapping from locations to variables, a lock environment t which is a mapping from pairs of locations and thread identifiers to states and a language term annotated by its thread identifier $n : e$. The basic rules of this relation are depicted in Fig. 6. For lack of space, we have omitted the propagation rules, which are straightforward.

In the relation we use an idiom that is standard in languages with linear values. Evaluation does not terminate with a value, but with a variable. This variable, called *auto-variable*, is automatically produced, but can be merged with program variables in a transparent way. Once a value v has been reached, it gets bound to a fresh variable z and placed in the store S. Access to this value may be given only through the fresh variable. We define in Fig. 5 a store lookup function $S; x \Downarrow S'; v$ which ensures that linear objects may be used by the program only once and the state x operator that returns

$$\boxed{S;\mu;t;n:e \hookrightarrow S';\mu';t';n:e'}$$

$$\frac{\text{fresh } z}{S;\mu;t;n:v \hookrightarrow S,z\mapsto v;\mu;t;n:z} \qquad \frac{S;w \Downarrow S';{}^{q}\lambda x:\tau.\,e}{S;\mu;t;n:w\ y \hookrightarrow S';\mu;t;n:e[x\mapsto y]} \qquad \frac{}{S;\mu;t;n:e\,[\rho] \hookrightarrow S;\mu;t;n:e}$$

$$\frac{\text{fresh } i}{S;\mu;t;n:\mathsf{new}\ x \hookrightarrow S;\mu,i\mapsto x;t;n:{}^{\mathsf{L}}\ulcorner i,{}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\ i,\mathsf{loc}\ i)\urcorner} \qquad \frac{\begin{array}{cc} S;w\Downarrow S_1;{}^{L}\ulcorner i,w_0\urcorner & S_1;w_0\Downarrow S_2;{}^{L}(w_1,w_2) \\ S_2;w_1\Downarrow S_3;{}^{\mathsf{L}}\mathsf{cap}\ i & S_3;w_2\Downarrow S_4;\mathsf{loc}\ i\end{array}}{S;\mu,i\mapsto z;t;n:\mathsf{free}\ w \hookrightarrow S_4;\mu;t;n:z}$$

$$\frac{\begin{array}{c} (i\times n,s)\in t \qquad\qquad T\sqsubseteq s\sqsubseteq R \\ S;w\Downarrow S_1;{}^{q}(w_1,w_2) \quad S_1;w_1\Downarrow S_2;{}^{q'}\mathsf{cap}\ i \quad S_2;w_2\Downarrow S_3;\mathsf{loc}\ i\end{array}}{S;\mu,i\mapsto z;t;n:\,!w \hookrightarrow S_3;\mu,i\mapsto z;t;n:z}$$

$$\frac{\begin{array}{c} (i\times n,T)\in t \qquad S;x\Downarrow S_1;{}^{q}(x_1,x_2) \\ S_1;x_1\Downarrow S_2;{}^{q'}\mathsf{cap}\ i \qquad S_2;x_2\Downarrow S_3;\mathsf{loc}\ i\end{array}}{S;\mu,i\mapsto z;t;n:x:=y \hookrightarrow S_3;\mu,i\mapsto y;t;n:\mathsf{unit}}$$

$$\frac{\begin{array}{c} \text{fresh } n_1 \quad \text{fresh } n_2 \quad t'=t\setminus\{i'\times n:s' \mid (\exists i's').[i'\times n:s'\in t]\} \\ \cup\{i\times n_1:s \mid ({}^{s\ \mathsf{at}\ \pi}\mathsf{cap}\ i\in cl_1)\}\cup\{i\times n_2:s \mid ({}^{s\ \mathsf{at}\ \pi}\mathsf{cap}\ i\in cl_2)\}\end{array}}{S;\mu;t;n:{}^{cl_1}e_1\ \|\ {}^{cl_2}e_2 \hookrightarrow S;\mu;t';n:n_1:e_1\#n_2:e_2}$$

$$\frac{S;\mu;t;n_1:e_1\hookrightarrow S';\mu';t';n_1:e_1'}{\begin{array}{c}S;\mu;t;n:n_1:e_1\#n_2:e_2\hookrightarrow \\ S';\mu';t';n:n_1:e_1'\#n_2:e_2\end{array}} \qquad \frac{S;\mu;t;n_2:e_2\hookrightarrow S';\mu';t';n_2:e_2'}{\begin{array}{c}S;\mu;t;n:n_1:e_1\#n_2:e_2\hookrightarrow \\ S';\mu';t';n:n_1:e_1\#n_2:e_2'\end{array}}$$

$$\frac{\begin{array}{c} t'=t\setminus\{i'\times n':s' \mid (i'\times n':s')\in t\wedge(n'=n_1\vee n'=n_2)\} \\ \cup\{i'\times n:s' \mid (n'=n_1\vee n'=n_2)\wedge(\exists i's').[i'\times n':s')\in t]\} \\ s=\mathsf{LUB}(\mathsf{state}\ v_1,\mathsf{state}\ v_2)\end{array}}{S,z_1\mapsto v_1,z_2\mapsto v_2;\mu;t;n:n_1:z_1\#n_2:z_2 \hookrightarrow S;\mu;t';n:{}^{s}(z_1,z_2)}$$

$$\frac{\text{fresh } \rho' \quad S;z\Downarrow S';{}^{q}\mathsf{cap}\ i \quad s=\mathsf{T}\Rightarrow(\nexists n's')[n\neq n'\wedge(i\times n':s')\in t] \quad s=\mathsf{R}\Rightarrow(\nexists n')[(i\times n':\mathsf{T})\in t]}{\begin{array}{c}S;\mu;t;n:\mathsf{lock}\ (x=z)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y=e_1\ \mathsf{in}\ e_2\hookrightarrow \\ S',z\mapsto{}^{s\ \mathsf{at}\ \rho'}\mathsf{cap}\ i;\mu;t,(i\times n,s);n:\mathsf{lock\$}\ (x=z)\ \mathsf{as}\ s\ \mathsf{at}\ \rho'\ \mathsf{then}\ y=e_1[\rho\mapsto\rho'][x\mapsto z]\ \mathsf{in}\ e_2\end{array}}$$

$$\frac{S;z\Downarrow S';{}^{q}\mathsf{cap}\ i \quad s=\mathsf{R}\Rightarrow(\exists n')[(i\times n':\mathsf{T})\in t] \quad s=\mathsf{T}\Rightarrow(\exists n's')[n\neq n'\wedge(i\times n':s')\in t]}{S;\mu;t;n:\mathsf{lock}\ (x=z)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y=e_1\ \mathsf{in}\ e_2\hookrightarrow S';\mu;t;n:\mathsf{lock}\ (x=z)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y=e_1\ \mathsf{in}\ e_2}$$

$$\frac{}{\begin{array}{c}S,z\mapsto{}^{s\ \mathsf{at}\ \rho}\mathsf{cap}\ i;\mu;t,(i\times n,s);n:\mathsf{lock\$}\ (x=z)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y=w\ \mathsf{in}\ e_2\hookrightarrow \\ S;\mu;t;n:e_2[y\mapsto w]\end{array}}$$

$$\frac{\text{fresh } \rho' \quad s\sqsubseteq\mathsf{R}\Rightarrow t'=t,i\times n,s \quad s=\mathsf{U}\Rightarrow t'=t}{\begin{array}{c}S,z\mapsto{}^{L}\mathsf{cap}\ i;\mu;t;n:\mathsf{let!}\ (x=z)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y=e_1\ \mathsf{in}\ e_2\hookrightarrow \\ S,z\mapsto{}^{s\ \mathsf{at}\ \rho'}\mathsf{cap}\ i;\mu;t';n:\mathsf{let\$}\ (x=z)\ \mathsf{as}\ s\ \mathsf{at}\ \rho'\ \mathsf{then}\ y=e_1[\rho\mapsto\rho'][x\mapsto z]\ \mathsf{in}\ e_2\end{array}}$$

$$\frac{s\sqsubseteq\mathsf{R}\Rightarrow t'=t\setminus\{i\times n,s\} \quad s=\mathsf{U}\Rightarrow t'=t}{S,z\mapsto{}^{s\ \mathsf{at}\ \rho}\mathsf{cap}\ i;\mu;t;n:\mathsf{let\$}\ (x=z)\ \mathsf{as}\ s\ \mathsf{at}\ \rho\ \mathsf{then}\ y=w\ \mathsf{in}\ e_2\hookrightarrow S,z\mapsto{}^{\mathsf{L}}\mathsf{cap}\ i;\mu;t';n:e_2[x\mapsto z][y\mapsto w]}$$

Fig. 6. Operational semantics.

the state part of the qualifier of a value v.

Memory μ in our semantic rules is used to handle the contents of references, in a standard way. Following our previous work [15], locations are bound to variables, instead of values, which comes naturally, given that evaluation in our language ends with variables. The actual value may be regained from the binding of the variable inside the store. In this way, we take linearity handling for the contents of references for free.

A parallel expression $n\ :\ {}^{cl_1}e_1\ \|\ {}^{cl_2}e_2$ is evaluated to an intermediate expression $n : n_1 : e_1\#n_2 : e_2$ attributing fresh thread identifiers to the new threads. At the same time, all locks with thread identifier n are removed from the lock set t and are passed to the child threads as dictated by the annotations cl_1 and cl_2. After non-deterministically evaluating both e_1 and e_2 to values, we restore the original lock environment and return a pair consisting of these two values with an appropriate qualifier.

Both expressions that manipulate states, let! and lock utilize two auxiliary expressions, not available to the user: let\$

and lock\$. These kind of expressions are needed for our metatheory, as shown in our previous work [15]. In the original expressions, e is evaluated to a capability and evaluation continues with the auxiliary expression. In the case of lock, evaluation continues only when the requested lock is available, in which case the lock and the corresponding capability are added to the set t and S respectively, for the evaluation of e_1, and are removed when this is finished. In the case of let!, the existing linear capability is removed from S in order to avoid having a linear and a non-linear capability for the same location at the same time. The new capability is added to S, and in case this is not an U capability, the appropriate lock is also added to t. When evaluation of e_1 is finished, the capability and the lock are purged and the linear capability is reinstated.

In the evaluation of lock $(x = e)$ as s at ρ then $y = e_1$ in e_2 we follow the spin-lock approach. That is, once e is evaluated to a capability for a location i, we check whether any other thread possesses a conflicting lock for i. In case the lock is not available, the expression evaluates to itself. Otherwise, the current thread is given the required lock, and proceeds as described above.

IV. Conclusion

In this paper we have presented a substructural type system, that can be used to detect memory violations and data races in a concurrent programming language. Our system is based on capabilities, which can be of four types: linear (exclusive access, non shareable), thread-exclusive (read and write access, shareable within a thread), read-only (read access, freely shareable), and unrestricted (no access, freely shareable). Two special language constructs, let! and lock can be used to change the types of capabilities in a safe and controlled way.

Acknowledgment

This research is partially funded by the programme for supporting basic research (ΠΕΒΕ 2010) of the National Technical University of Athens under a project titled "Safety properties for concurrent programming languages."

References

[1] H. G. Baker, "Lively linear Lisp: Look ma, no garbage!" *ACM SIGPLAN Notices*, vol. 27, no. 8, pp. 89–98, Aug. 1992.

[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*, Nov. 2002, pp. 211–230.

[3] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002, pp. 258–269.

[4] R. Ennals, R. Sharp, and A. Mycroft, "Linear types for packet processing," in *Proceedings of the 13th European Symposium on Programming*. Springer, 2004, pp. 204–218.

[5] C. Flanagan and S. N. Freund, "Type inference against races," in *Proceedings of the International Symposium on Static Analysis*. Springer-Verlag, 2004, pp. 116–132.

[6] P. Gerakios, N. Papaspyrou, and K. Sagonas, "Race-free and memory-safe multithreading: design and implementation in Cyclone," in *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, 2010, pp. 15–26.

[7] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, pp. 1–102, 1987.

[8] 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*, 2002, pp. 282–293.

[9] M. Hicks, G. Morrisett, D. Grossman, and T. Jim, "Safe and flexible memory management in Cyclone," University of Maryland, Department of Computer Science, Tech. Rep. CS-TR-4514, Jul. 2003.

[10] H. Huang, L. Wittie, and C. Hawblitzel, "Formal properties of linear memory types," Dartmouth College, Computer Science, Hanover, NH, Tech. Rep. TR2003-468, August 2003.

[11] N. Kobayashi, "Type systems for concurrent programs," in *Proceedings of 10th Anniversary Colloquium of UNU/IIST*, ser. LNCS, vol. 2757. Springer, 2003, pp. 439–453.

[12] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.

[13] G. Morrisett, A. Ahmed, and M. Fluet, "A linear language with locations," in *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, 2005, pp. 293–307.

[14] M. Odersky, "Observers for linear types," in *Proceedings of the 4th European Symposium on Programming*. Springer-Verlag, 1992, pp. 390–407.

[15] M. A. Papakyriakou and N. S. Papaspyrou, "From linear to unrestricted and back: Type safety and the let-bang construct," 2010, unpublished manuscript, School of Electrical and Computer Engineering, National Technical University of Athens, Greece.

[16] M. C. Rinard, "Analysis of multithreaded programs," in *Proceedings of the International Symposium on Static Analysis*, 2001, pp. 1–19.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," *ACM Transactions on Computer Systems*, vol. 15, 1997.

[18] R. Shi and H. Xi, "A linear type system for multicore programming," in *Proceedings of the 13th Brazilian Symposium on Programming Languages*, August 2009.

[19] R. Shi, D. Zhu, and H. Xi, "A modality for safe resource sharing and code reentrancy," in *Proceedings of International Colloqium on Theoretical Aspects of Computing*, ser. LNCS, vol. 6255. Natal, Brazil: Springer-Verlag, September 2010, pp. 382–396.

[20] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim, "Safe manual memory management in Cyclone," *Science of Computer Programming*, vol. 62, no. 2, pp. 122–144, 2006.

[21] P. Wadler, "Linear types can change the world!" in *Programming Concepts and Methods*, M. Broy and C. Jones, Eds. Amsterdam: North Holland, 1990, pp. 347–359.

[22] D. Walker, "Substructural type systems," in *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. The MIT Press, 2005, ch. 1.

[23] D. Walker and K. Watkins, "On regions and linear types," in *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, 2001, pp. 181–192.

[24] L. Wittie and J. Lockhart, "Type-safe concurrent resource sharing," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 8, pp. 767–795, 2011.