Safe Reactive Programming: The FunLoft Proposal

Frédéric Boussinot, Frederic Dabrowski

Safe Reactive Programming: The FunLoft Proposal

Frédéric Boussinot[1] and Frédéric Dabrowski[2]

[1] INRIA, B.P. 93, 06902 Sophia-Antipolis Cedex, France
frederic.boussinot@inria.fr
[2] IRISA, Campus de Beaulieu, 35 042 Rennes Cedex, France
frederic.dabrowski@irisa.fr

Abstract. We propose a multicore-ready programming language based on a two-level shared memory model. Concurrency units are schedulers and threads which are dispatched on available cores in a preemptive way. Each scheduler is in charge of its own portion of the memory. At runtime, several threads may link to a common scheduler. In this case, they enter a cooperative mode, evolve in synchronous rounds, and are granted access to the scheduler memory. At the opposite, an autonomous thread runs at its own pace but has access only to a local memory. The language ensures that programs are free of memory leaks, that code between two cooperation points is atomic, and that rounds are fair and always terminate (no run-time error nor divergence).

1 Introduction

Back in 1960, [14] presents concurrent programming as a major revolution in computer programming. It already points out unexpected concurrent accesses to shared memory as one of the most difficult problems to face. At the language level, appropriate synchronization constructs may help to build reliable software. Such one construct is the now well-known *monitor* of Brinch Hansen and Hoare. Quoting [23]:

> A primary task [...] is to construct resource allocation (or scheduling) algorithms for resources of various kind [...] Each scheduler will consist of a certain amount of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources. Such a collection of associated data and procedures is known as a monitor [...]

Multi-threading concurrent programming maps in a very natural way onto multicore architectures. Execution threads are mapped onto available cores and access a common global memory. It is now widely recognized that programming with low-level synchronization mechanisms such as locks is a hard and error prone task. A possible alternative is the use of processes which are concurrent units with separate memory and run-time protection. Processes have been introduced

in many concurrent languages as denoting concurrent programs with private memory and structured communication means (for example, in Erlang, Occam, or CML). As opposed to processes, threads do not have built-in memory protection and have direct access to a shared memory. A crucial point is that threads are more efficient than processes, quoting [10]

> [...] threads are an attractive tool for allowing a program to take advantage of the available hardware. The alternative, with most conventional operating systems, is to configure your program as multiple separate processes, running in separate address spaces. This tends to be expensive to set up, and the costs of communicating between address spaces are often high, even in the presence of shared segments. By using a lightweight multi-threading facility, the programmer can utilize the processors cheaply.

The language we propose, namely FunLoft, relies on a two-level shared memory model. Concurrency units are schedulers and threads dispatched on available cores, in a preemptive way. Informally, at a first glance, one can see schedulers and threads as processes, in the sense that each of them is in charge of its own portion of the memory. At runtime, several threads may link to a common scheduler. In this case, they enter a cooperative mode, evolve in synchronous rounds and are granted access to the scheduler memory. In particular, in this second level, one can see threads as standard threads running in the memory space of a process, except that they execute cooperatively and have access to specific synchronization constructs. When autonomous, a thread runs at its own pace but only has access to its local memory. To guarantee that accesses to memory locations always respect the above-mentioned discipline, we rely on a simple type and effect system mapping memory locations to schedulers and threads and checking access rights. Moreover, we note that, ideally, a language for multi-threaded concurrent programming should meet at least the following requirements:

1. it should have a formal semantics, as intuitive as possible.
2. it should propose high-level constructs for communication and synchronization and these constructs should be efficient, even when there is a large number of concurrent behaviors.
3. it should propose means to define sections of code that should be atomic.
4. it should be safe, in the sense that no run-time error should occur.
5. it should be able to benefit from multi-core architectures.

We propose a language aiming at fulfilling the above-mentioned requirements. In particular, the design of the language ensures that : programs are free of memory leaks, code between two cooperation points is atomic and rounds are fair and always terminate (no run-time error nor divergence).

FunLoft[1], is presented in Section 2. The language is informally described in Section 2.1 and static analyses are considered in Section 2.2. Section 3 describes an example of colliding particles that illustrates the use of multi-core machines. Related work is considered in Section 4, and Section 5 concludes the paper.

2 FunLoft

FunLoft is an experimental programming language concerned with safety, and qualitative and quantitative resource control in a concurrent setting. Here, safety basically means absence of run-time errors. Resources that are intended to be controlled are the memory and the processing units. Qualitative resource control means control of accesses to the shared memory (see below). Quantitative resource control means absence of memory leaks.

The objective of FunLoft is however larger than to combine concurrency, resource control, and safety: first, the language proposes a particular form of concurrency, as an alternative to standard concurrency frameworks based on preemptive threads (Java threads or Pthreads [26]). Second, FunLoft allows users to benefit from real parallelism (in particular, the one offered by multi-core machines) when programming a single application. Third, efficiency is an important concern, which was present all along the language design.

2.1 Overview of FunLoft

A program in FunLoft is a list of definitions of variables, types, functions, and modules (see below). In order to be executed, a program must contain the definition of a module named `main`, which is the program entry-point. A variable is a name to which a value is associated. Values are first-order only and range over standard values (booleans and integers, for example), values of user-defined types, and thread, event, and scheduler names. Type definitions introduce polymorphic structured datatypes, possibly recursive (e.g. lists), built from union and concatenation of other types. Functions can be recursively defined; however, recursion is checked for termination, in a sense explained later, and thus function calls always terminate.

Schedulers A scheduler controls the threads which are linked to it and provides them with a portion of shared memory. A special scheduler (the *implicit scheduler*) is automatically launched by each executable program and a thread created as an instance of the main module is linked to it. The basic task of a scheduler is to schedule the threads which are linked to it. The scheduling is cooperative: linked threads have to return the control to the scheduler, in order to let the other threads execute. Leaving the control can be either explicit, via the instruction `cooperate`, or implicit, by waiting for an event (`await` statement) which is not present. All linked threads are cyclically considered by the scheduler until all of them have reached a suspension point (`cooperate` or `await`). Then, and only then, a new cycle can start. Cycles are called *instants*. A scheduler thus defines an automatic synchronization mechanism which forces all the threads it controls to run at the same pace: all the threads must have finished their execution for the current instant before the next instant can start. Note that the same thread can receive the control several times during the same instant; this is for example the case when the thread waits for an event which is generated (`generate` statement) by another thread later in the same instant. In this case, the thread

receives the control a first time and then suspends, waiting for the event. The control then goes to the other threads, and returns back to the first thread after the generation of the event. At creation, each thread is linked to one scheduler (by default, the implicit scheduler). Several schedulers can be defined and simultaneously run in the same program. Schedulers thus define synchronous areas in which threads execute in cooperation. Basically, schedulers run autonomously, in a preemptive way, under the supervision of the OS.

During their execution, threads can unlink from the scheduler to which they are currently linked (`unlink` statement), and become free from any scheduler synchronization. Such free unlinked threads are, like schedulers, run by kernel threads under the supervision of the OS. Threads can also dynamically move from a scheduler to another scheduler (`link` statement).

Modules Modules are templates from which threads, called *instances*, are created. A module can have parameters which define corresponding parameters of its instances. A module also defines variables local to each created thread. As opposed to functions, modules cannot be recursively defined. The body of a module is basically a sequence of instructions with usual control statements such as `loop`, `repeat` statements, and conditional statements. There are two types of instructions: atomic instructions and non-atomic ones. Atomic instructions are logically run in a single instant. Function calls belong to this kind of instruction. Execution of non-atomic instructions may need several instants to complete. This is typically the case of the instruction `await`.

Communication and Synchronisation The simplest way for threads to communicate is of course to use shared variables. For example, a thread can set a boolean variable to indicate that a condition is true, and other threads can test the variable to know the status of the condition. This basic pattern works well when all the threads accessing the variable are linked to the same scheduler. Indeed, in this case atomicity of the accesses to the variable is guaranteed by the cooperativeness of the scheduler. A general way to protect a data from concurrent accesses is thus to associate it with a scheduler to which threads willing to access the data should first link to[3]. Events are basically used by threads to avoid busy-waiting on conditions. An event is always associated with a scheduler (by default, the implicit scheduler) which is in charge of it during all its lifetime. An event is either present or absent during each instant of the scheduler which manages it. It is present if it is generated by the scheduler (e.g. as a consequence of interactions with the environment) at the beginning of the instant, or if it is generated by one of the threads executed by the scheduler during the instant; it is absent otherwise. The presence or the absence of an event can change from an instant to another, but all threads always "see" the presence or absence of events in the same way, independently of the order in which the threads are scheduled. For this reason, we say that events are *broadcast*. Values can be associated with

[3] Note the analogy with monitors, described indeed by Hoare as schedulers, in the citation of Section 1.

event generations; they are collected during each instant (`get_all_values` statement) and their collection becomes available as a list at the next instant.

It is possible to define *synchronised schedulers* that are run autonomously but that share the same instants. Events are shared by synchronised schedulers, which is possible because synchronised schedulers themselves share instants. Thus, if `s1` and `s2` are two synchronised schedulers, it is for example possible to wait for an event in `s1` and to generate the same event in `s2`.

Memory Model The global memory is divided in several parts:

- A private memory for each thread. This memory is initialised when the thread is created, from the parameters and the local variables of the module from which the thread is created. The system checks that no other thread can have access to this private memory.
- A private memory for each scheduler. This memory can only be accessed by the threads linked to the scheduler. The system checks that it is not accessible from unlinked threads, nor from threads linked to another scheduler.

Thus, there exists no global variable shared by distinct schedulers, and a variable that is shared by several threads belongs to the unique scheduler that controls these threads and cannot be accessed by other threads. This is the way data-races are ruled-out in FunLoft.

2.2 Formalisation

Despite its two concurrency levels, FunLoft has a formal semantics which is very concise. A semantics restricted to a unique scheduler is presented in [17]; a complete semantics, when there are several schedulers, is described in [16] for a language very close to FunLoft, and the full formalisation of Funloft is described in a technical paper [12]. Among the properties checked at compile-time in FunLoft programs, are the following:

1. the program is well-typed under standard definitions
2. functions always terminate and linked threads always cooperate (despite functions are recursive and threads may never terminate).
3. The amount of memory used by the program and the number of (simultaneously) living threads are bound by a function of the program inputs (despite the use of complex datatypes and the dynamic creation of threads).
4. the memory model described above is actually respected; more precisely the compiler checks that there exists a mapping from memory locations to schedulers and threads such that any execution respects the corresponding access rights.

In particular, (1), (2) and (3) ensure that a thread can neither produce a run-time error, nor prevent the termination of instants, and thus delay endlessly the handling of further inputs. Thus, a thread can never prevent other threads from progressing.

Property (4) ensures that, when linked to a scheduler, a thread always executes atomically (at least from a logical point of view) the portions of code between two cooperation points.

Details on the formalisation of FunLoft are out of the scope of this paper. We refer the interested reader to [16, 12]. Ongoing work on computing explicit bounds on the amount of memory used by the program can also be found in [7] (in FunLoft, we only check for the existence of bounds, and we do not compute them explicitly). However, here are some hints about the static analyses performed by the compiler.

Basically, the compiler relies on a type and effect system mapping disjoint subsets of memory locations to owners (schedulers and threads). It enforces a logical separation of the memory as follows:

1. when autonomous, a thread can only access its own memory,
2. when linked to a scheduler, a thread can only access its own memory and the memory of the scheduler it is linked to.

As linked threads behave cooperatively, it is immediate that the execution of code between two cooperation points is atomic.

Termination of recursive functions is controled. In the present version of FunLoft, recursivity can only concern parameters of inductive types. The size of a parameter `p` is smaller than the size of a parameter `q` if `p` is a sub-term of `q`. For lists of parameters, one extends "lexicographically" the notion of size. In a complete sequence of calls (starting and ending on calls of the same function), one checks that at each step parameters are strictly decreasing. Thus, any function call is forced to terminate after a finite number of recursive calls.

To control the size of the global memory used by a program, memory locations are stratified over a finite domain. We rely on a simple data flow analysis to ensure that no cyclic dependence is possible for memory locations carrying complex datatypes (simple datatypes such as integers do not have this restriction). A value put into a memory location is always the result of the composition of a finite number of functions applied to values read from upper memory locations. As functions always terminate, the size of values held at a memory location is bound by a function of the size of the program inputs.

Finally, we rely on a simple static analysis to guarantee that the number of living threads remains bounded. Actually, we avoid (asynchronous) thread creation in loops. Then, at any point of execution, the number of living threads only depends on the size of the program inputs. Thanks to the design of the language, at the beginning of each instant, the size of the stack associated with a thread is bound by a constant. This entails that (given the existence of a garbage collection mechanism) the size of the global memory used by a program is a function of the program inputs.

Several other static controls are also made. For example, an event cannot be generated by a thread linked to a scheduler, and waited for by a thread linked to another scheduler, unless the two schedulers are synchronised (otherwise, a buffer to store generations would be necessary, and, as the schedulers would be asynchronous, the buffer size could not be bounded).

2.3 Implementation

We are currently developing an implementation prototype, available at [1]. Autonomous unlinked threads and schedulers co-existing in the same application are run by distinct kernel processing units (kernel threads) on distinct processors (multi-processors or multi-core architectures).

This prototype is used to implement several examples of graphical simulations, including cellular automata made of several thousands of cells, each cell being a thread. The simulation presented in Section 3 is one of these examples. These examples show, in particular, that systems with large numbers of concurrent programs can be efficiently implemented in FunLoft.

3 Colliding Particles

One considers the simulation of colliding particles. A simple brute-force algorithm is used in which each particle considers all other particles (leading thus to a complexity square in the number of particles). The particles are partitioned in two sets of equal size, and each set is linked to a distinct scheduler. In order to make the collisions realistic, the two schedulers are synchronised.

3.1 Collision Processing

A particle is made of four references on floats and of a constant color. The data-type `particle_t` is defined by:

```
type particle_t = Particle of
      float ref      * // x coord
      float ref      * // y coord
      float ref      * // x speed
      float ref      * // y speed
      color_t          // color
```

The `process_all_collisions` function processes collisions between a particle given as first parameter, and a list of particles given as second parameter (collision between two particles is performed by the function `collide` not described here for simplicity). The parametric definition of the inductive type of lists is:

```
type 'a list = Nil_list | Cons_list of 'a * 'a list
```

The recursive definition of `process_all_collisions` is:

```
let process_all_collisions (me,list) =
  match list with
      Nil_list -> ()
    | Cons_list (other,tail) ->
        begin
```

```
        collide (me,other);
        process_all_collisions (me,tail);
      end
  end
```

The system verifies that the function always terminates. This results from the fact that the second parameter (`tail`) of the recursive call is smaller than (is a subterm of) the second parameter (`list`) of the initial call (the first parameter remaining unchanged).

3.2 Particle Behaviour

At each instant, a particle generates the `collide_event` with an associated value containing its coordinates, and then collects all the values generated on this very event. Thus, at the next instant, the coordinates of all the particles have been collected. They are then processed, and the particle is finally moved according to inertia. The code of module `collide_behavior` is:

```
let module collide_behavior (me,collide_event) =
  let r = ref Nil_list in
    loop
      begin
        generate collide_event with particle2coord (me);
        get_all_values collide_event in r;
        process_all_collisions (me,!r);
        inertia (me);
      end
```

The system checks that the body of the `loop` statement, which defines a cyclic behavior, cannot terminate instantly. Otherwise, the execution of an instance of the module would not cooperate and would prevent all other threads from execution. The absence of immediate termination comes from the fact that the collection of all the values takes the whole instant; thus, the processing of the collected values is delayed to the next instant.

The particle behavior is actually made of three threads: one for bouncing on the borders of the simulation (`bounce_behavior`), one for colliding with other particles (`collide_behavior`), and one for drawing the particle on the screen (`draw_behavior`):

```
let module particle_behavior (collide_event,color) =
  let s = new_particle (color) in
    begin
      thread bounce_behavior (s);
      thread collide_behavior (s,collide_event);
      thread draw_behavior (s);
    end
```

Note that the particle `s` (created by the function `new_particle` in charge of randomly placing the particle on the screen) is shared between the three threads.

3.3 Simulation

First, two schedulers `s1` and `s2` are declared at top-level (keyword `scheduler`). The presence of the keyword `and` which links the two declarations makes the two schedulers synchronised. Then, two events `draw_event` and `collide_event` are declared (keyword `event`). The system will infer that these events are associated to the two schedulers `s1` and `s2`. Then, after linking to scheduler `s1`, a thread instance of the module `graphics` and one of the module `draw_processor` are created and linked to `s1`. In the same way, half of the particles are created and linked to `s1`. Finally, after linking to `s2`, the other half of particles are created and linked to `s2`. The `main` module is:

```
let s1 = scheduler
and s2 = scheduler

let module main () =
  let draw_event = event in
  let collide_event = event in
    begin
      link s1 do begin
         thread graphics (maxx,maxy,BLACK);
         thread draw_processor (draw_event);
         repeat particle_number / 2 do
           thread particle_behavior (collide_event,draw_event,GREEN);
      end;
      link s2 do
         repeat particle_number / 2 do
           thread particle_behavior (collide_event,draw_event,RED);
    end
```

The compiler verifies that no data-race can appear due to the sharing of particles: a particle is only shared by threads linked to the same scheduler. It also verifies that the two events `draw_event` and `collide_events` are always used by threads linked to `s1` or to `s2`.

3.4 Execution Results

The machine characteristics are: a Mac running OS X 10.4.10, processor Intel Core 2 Duo, 2.33 GHz, 2GB of memory. Graphics is based on SDL[3].

The simulation contains 500 particles. The CPU usage is shown in the right part of Figure 1 (the graphical window is masked during the measure). This is to compare with the usage obtained with the single threaded version of the simulation, shown in the left part.

The time for simulating 100 instants (one instant corresponds to the execution of all the particles) is shown on the left part of Figure 2 (representing the output of the unix command `time`), with a memory footprint of about 70MB. Note that the total number of performed interactions is about $100 * 500^2 = 25^6$. The case of 1000 particles is shown on the right of Figure 2.

X

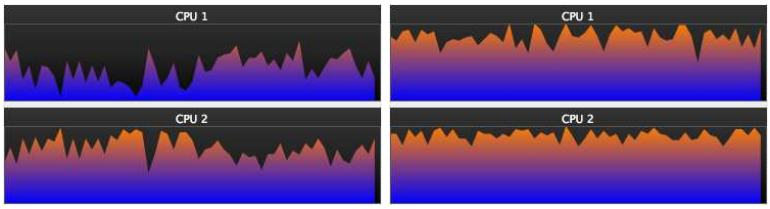

Fig. 1. CPU usage. Left: 1 scheduler. Right: 2 schedulers

500 particles	1 sched	2 scheds
real	0m21.832s	0m14.189s
user	0m21.102s	0m21.369s
sys	0m0.220s	0m0.379s

1K particles	1 sched	2 scheds
real	1m20.564s	0m53.724s
user	1m19.587s	1m23.508s
sys	0m0.491s	0m1.078s

Fig. 2. Time for 500 particles (left) and 1000 particles (right) during 100 instants

These results show the benefit from using a multi-core computer on a simulation in with an heavy load of communication between threads.

4 Related work

FunLoft is strongly connected to synchronous languages [22]. In these languages, as in FunLoft, concurrent behaviors share a logical clock, the units of which are called "instants". Concurrent behaviors automatically synchronise at each instant and communicate using broadcast signals to which data can be associated. At each instant, a signal is consistently seen as present or absent by all the parallel processes, which also read the same associated data. In Esterel [9], one of the first synchronous languages, reactivity (termination of the instants) and determinism are guaranteed for the kernel of the language. However, this kernel has deep constraints that restrict the language to applications such as the production of circuits.

Strongly related to synchronous languages is a family of formalisms grouped under the name "Reactive Programming" [2]. Reactive programming is implemented in various languages (SML, Java, ML, C, and Scheme). The formalism of FairThreads [11], built over C, deserves a special status amongst the reactive programming formalisms for two reasons: first, in FairThreads, concurrent programs are basically threads, while the other formalisms use parallel composition, adapted for instants; second, FairThreads considers a mix of cooperative and preemptive threads, which gives a model in which synchrony and asynchrony are both present. FairThreads aims at being a more general purpose programming language than its ancestors, but loses reactivity and determinism. The FunLoft language is very close to FairThreads as regards the concurrency model. However, although the FairThreads documentation suggests that schedulers might

be used to access shared memory in a cooperative way, no static checks ensure that threads not linked to a given scheduler (possibly linked to other schedulers) do not interfere with the threads linked to this scheduler.

There exists a line of experimental work aiming at obtaining both language expressivity and strong properties such as reactivity and determinism. For reactivity, we refer to [7] for results in a synchronous π-calculus[5]. In this paper, a static analysis is defined that guarantees that instants always terminate in time polynomial in the size of the program inputs. It extends preliminary results that can be found in [8, 6]. For determinism, we refer to [4] for results applying to the same calculus. However, the synchronous π-calculus does not consider shared memory: in it, valued signals are the only means of communication between concurrent behaviors.

The approach followed in FunLoft is very close to the work on static analysis for data races detection [15, 18, 19, 13, 25, 24] and to the work on checking for atomicity in a lock-based concurrent paradigm [21, 20]. Our point of view is that locks are a too low-level tool for general purpose concurrent programming. We hope that the programming model of FunLoft will provide a higher-level model of concurrency, thus helping in developing and debugging applications.

5 Conclusion

At the implementation level, threads allow users to benefit from multi-core architectures and from shared memory. At the logical level, one needs to ensure atomic execution of code blocks. We propose a framework based on FairThreads in which static checks insure three fundamental properties: that programs are free of memory leaks, that code between two cooperation points is atomic, and that rounds are fair and always terminate (no run-time error nor divergence). In this framework, fair threads can be as efficient as standard preemptive threads, but as secure as processes. The multi-scheduler aspect of the proposal fits well with multi-core architectures: schedulers and unlinked threads are executed by kernel threads that can be securely run in parallel by distinct cores without needing dynamic memory protection mechanisms.

References

1. FunLoft: http://www-sop.inria.fr/mimosa/rp/FunLoft.
2. Reactive Programming: http://www-sop.inria.fr/mimosa/rp.
3. Simple Directmedia Layer: http://www.libsdl.org.
4. R.M. Amadio and M. Dogguy. Determinacy in a Synchronous pi-Calculus, 2007. hal-00159764.
5. Roberto M. Amadio. A Synchronous pi-Calculus. *Journal of Information and Computation*, 205(9):1470–1490, 2007.
6. Roberto M. Amadio and Frédéric Dabrowski. Feasible Reactivity for Synchronous Cooperative Threads. *Electronic Notes in TCS*, 154(3):33–43, 2005.

7. Roberto M. Amadio and Frédéric Dabrowski. Feasible Reactivity in a Synchronous pi-Calculus. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles and practice of declarative programming*, pages 221–230, New York, NY, USA, 2007. ACM Press.
8. Roberto M. Amadio and Silvano Dal Zilio. Resource Control for Synchronous Cooperative Threads. *Theoretical Computer Science*, 358:229–254, 2004.
9. G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.
10. A.D. Birrell. An Introduction to Programming with Threads. *Digital Report*, (35), 1989.
11. F. Boussinot. FairThreads: Mixing Cooperative and Preemptive Threads in C. *Concurrency and Computation: Practice and Experience*, 18:445–469, 2005.
12. F. Boussinot and F. Dabrowski. Formalisation of FunLoft. *Research Report*, 2007.
13. Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-free Java Programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 56–69, New York, NY, USA, 2001. ACM Press.
14. P. Brinch Hansen. *The Origin of Concurrent Programming*. Springer, 2002.
15. Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 91–108, London, UK, 1999. Springer-Verlag.
16. F. Dabrowski. *Programmation Réactive Synchrone, Langage et Contrôle des Ressources*. Phd, University Paris 7, Department of Computer Science, 2007.
17. F. Dabrowski and F. Boussinot. Cooperative Threads and Preemptive Computations. In *Proc. of TV'06 – Multithreading in Hardware and Software: Formal Approaches to Design and Verification*. Seattle, 2006.
18. Cormac Flanagan and Stephen N. Freund. Type-based Race Detection for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.
19. Cormac Flanagan and Stephen N. Freund. Type Inference Against Races. *Sci. Comput. Program.*, 64(1):140–165, 2007.
20. Cormac Flanagan and Shaz Qadeer. A Type and Effect System for Atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.
21. Cormac Flanagan and Shaz Qadeer. Types for Atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM Press.
22. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, New York, 1993.
23. C.A.R. Hoare. Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.
24. Mayur Naik and Alex Aiken. Conditional Must not Aliasing for Static Race Detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 327–338, New York, NY, USA, 2007. ACM Press.
25. Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. *SIGPLAN Not.*, 41(6):308–319, 2006.
26. B. Nichols, D. Buttlar, and Proulx Farrell J. *Pthreads Programming*. O'Reilly, 1996.