Received XXXXX; | Revised XXXXX; | Accepted XXXXX

DOI: xxx/xxxx

RESEARCH ARTICLE

Concurrency in C∀

Thierry Delisle | Peter A. Buhr*

[1]Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada

Correspondence
*Peter A. Buhr, Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, Canada. Email: pabuhr@uwaterloo.ca

Funding information
Natural Sciences and Engineering Research Council of Canada

Summary

C∀ is a modern, polymorphic, *non-object-oriented* extension of the C programming language. This paper discusses the design of the concurrency and parallelism features in C∀, and its concurrent runtime-system. These features are created from scratch as ISO C lacks concurrency, relying largely on the pthreads library for concurrency. Coroutines and lightweight (user) threads are introduced into C∀; as well, monitors are added as a high-level mechanism for mutual exclusion and synchronization. A unique contribution of this work is allowing multiple monitors to be safely acquired *simultaneously*. All features respect the expectations of C programmers, while being fully integrate with the C∀ polymorphic type-system and other language features. Experimental results show comparable performance of the new features with similar mechanisms in other concurrent programming-languages.

KEYWORDS
concurrency, parallelism, coroutines, threads, monitors, runtime, C, Cforall

1 | INTRODUCTION

This paper provides a minimal concurrency *Application Program Interface* (API) that is simple, efficient and can be used to build other concurrency features. While the simplest concurrency system is a thread and a lock, this low-level approach is hard to master. An easier approach for programmers is to support higher-level constructs as the basis of concurrency. Indeed, for highly-productive concurrent-programming, high-level approaches are much more popular[1]. Examples of high-level approaches are jobs (thread pool)[2], implicit threading[3], monitors[4], channels[5,6], and message passing[7,8].

The following terminology is used. A *thread* is a fundamental unit of execution that runs a sequence of code and requires a stack to maintain state. Multiple simultaneous threads give rise to *concurrency*, which requires locking to ensure access to shared data and safe communication. *Locking*, and by extension *locks*, are defined as a mechanism to prevent progress of threads to provide safety. *Parallelism* is running multiple threads simultaneously. Parallelism implies *actual* simultaneous execution, where concurrency only requires *apparent* simultaneous execution. As such, parallelism only affects performance, which is observed through differences in space and/or time at runtime. Hence, there are two problems to be solved: concurrency and parallelism. While these two concepts are often combined, they are distinct, requiring different tools[9, § 2]. Concurrency tools handle mutual exclusion and synchronization, while parallelism tools handle performance, cost, and resource utilization.

The proposed concurrency API is implemented in a dialect of C, called C∀ (pronounced C-for-all). The paper discusses how the language features are added to the C∀ translator with respect to parsing, semantics, and type checking, and the corresponding high-performance runtime-library to implement the concurrent features.

2 | C∀ OVERVIEW

The following is a quick introduction to the C∀ language, specifically tailored to the features needed to support concurrency. Extended versions and explanation of the following code examples are available at the C∀ website[10] or in Moss *et al.*[11].

C∀ is a non-object-oriented extension of ISO-C, and hence, supports all C paradigms. Like C, the building blocks of C∀ are structures and routines. Virtually all of the code generated by the C∀ translator respects C memory layouts and calling conventions. While C∀ is not object oriented, lacking the concept of a receiver (*eg*, this) and nominal inheritance-relationships, C has a notion of objects: "region of data storage in the execution environment, the contents of which can represent values"[12, 3.15]. While some object-oriented features appear in C∀, they are independent capabilities, allowing C∀ to adopt them while maintaining a procedural paradigm.

2.1 | References

C∀ provides multi-level rebindable references, as an alternative to pointers, which significantly reduces syntactic noise.

```
int x = 1, y = 2, z = 3;
int * p1 = &x, ** p2 = &p1,  *** p3 = &p2,          // pointers to x
    & r1 = x,   && r2 = r1,   &&& r3 = r2;           // references to x
int * p4 = &z, & r4 = z;

*p1 = 3; **p2 = 3; ***p3 = 3;       // change x
r1 =  3;     r2 = 3;      r3 = 3;       // change x: implicit dereferences *r1, **r2, ***r3
**p3 = &y; *p3 = &p4;               // change p1, p2
&r3 = &y; &&r3 = &&r4;              // change r1, r2: cancel implicit dereferences (&*)**r3, (&(&*)*)*r3, &(&*)r4
```

A reference is a handle to an object, like a pointer, but is automatically dereferenced the specified number of levels. Referencing (address-of &) a reference variable cancels one of the implicit dereferences, until there are no more implicit references, after which normal expression behaviour applies.

2.2 | with Statement

Heterogeneous data is aggregated into a structure/union. To reduce syntactic noise, C∀ provides a **with** statement (see Pascal[13, § 4.F]) to elide aggregate field-qualification by opening a scope containing the field identifiers.

```
struct S { char c; int i; double d; };
struct T { double m, n; };
// multiple aggregate parameters
void f( S & s, T & t ) {           void f( S & s, T & t ) with ( s, t ) {
    s.c; s.i; s.d;                     c; i; d;        // no qualification
    t.m; t.n;                          m; n;
}                                  }
```

Object-oriented programming languages only provide implicit qualification for the receiver.

In detail, the **with**-statement syntax is:

with-statement:
    `'with' '('` *expression-list* `')'` *compound-statement*

and may appear as the body of a routine or nested within a routine body. Each expression in the expression-list provides a type and object. The type must be an aggregate type. (Enumerations are already opened.) The object is the implicit qualifier for the open structure-fields. All expressions in the expression list are opened in parallel within the compound statement, which is different from Pascal, which nests the openings from left to right.

2.3 | Overloading

C∀ maximizes the ability to reuse names via overloading to aggressively address the naming problem. Both variables and routines may be overloaded, where selection is based on number and types of returns and arguments (as in Ada[14]).

```
// selection based on type
const short int MIN = -32768;                      |  short int si = MIN;
const int MIN = -2147483648;                       |  int i = MIN;
const long int MIN = -9223372036854775808L;        |  long int li = MIN;
// selection based on type and number of parameters
void f( void );                 |  f();
void f( char );                 |  f( 'a' );
void f( int, double );          |  f( 3, 5.2 );
// selection based on type and number of returns
char f( int );                  |  char c = f( 3 );
double f( int );                |  double d = f( 3 );
[char, double] f( int );        |  [d, c] = f( 3 );
```

Overloading is important for C∀ concurrency since the runtime system relies on creating different types to represent concurrency objects. Therefore, overloading eliminates long prefixes and other naming conventions to prevent name clashes. As seen in Section 3, routine main is heavily overloaded. As another example, variable overloading is useful in the parallel semantics of the **with** statement for fields with the same name:

```
struct S { int i; int j; double m; } s;
struct T { int i; int k; int m; } t;
with ( s, t ) {
    j + k;                               // unambiguous, s.j + t.k
    m = 5.0;                             // unambiguous, s.m = 5.0
    m = 1;                               // unambiguous, t.m = 1
    int a = m;                           // unambiguous, a = t.m
    double b = m;                        // unambiguous, b = s.m
    int c = s.i + t.i;                   // unambiguous, qualification
    (double)m;                           // unambiguous, cast s.m
}
```

For parallel semantics, both s.i and t.i are visible with the same type, so only i is ambiguous without qualification.

2.4 | Operators

Overloading also extends to operators. Operator-overloading syntax creates a routine name with an operator symbol and question marks for the operands:

```
int ++?(int op);                  // unary prefix increment    |  struct S { int i, j; };
int ?++(int op);                  // unary postfix increment   |  S ?+?( S op1, S op2) { // add two structures
int ?+?(int op1, int op2);        // binary plus               |      return (S){op1.i + op2.i, op1.j + op2.j};
int ?<=?(int op1, int op2);       // binary less than          |  }
int ?=? (int & op1, int op2);     // binary assignment         |  S s1 = {1, 2}, s2 = {2, 3}, s3;
int ?+=?(int & op1, int op2);     // binary plus-assignment    |  s3 = s1 + s2;       // compute sum: s3 == {2, 5}
```

2.5 | Constructors / Destructors

Object lifetime is a challenge in non-managed programming languages. C∀ responds with C++-like constructors and destructors, using a different operator-overloading syntax.

```
struct VLA { int len, * data; };                          // variable length array of integers
void ?{}( VLA & vla ) with ( vla ) { len = 10;  data = alloc( len ); }  // default constructor
void ?{}( VLA & vla, int size, char fill ) with ( vla ) { len = size;  data = alloc( len, fill ); } // initialization
void ?{}( VLA & vla, VLA other ) { vla.len = other.len;  vla.data = other.data; } // copy, shallow
void ^?{}( VLA & vla ) with ( vla ) { free( data ); }     // destructor
{
    VLA  x,         y = { 20, 0x01 },      z = y;         // z points to y
    //   x{};       y{ 20, 0x01 };         z{ z, y };     implicit calls
    ^x{};                                                 // deallocate x
    x{};                                      // reallocate x
    z{ 5, 0xff };                             // reallocate z, not pointing to y
    ^y{};                                     // deallocate y
    y{ x };                                   // reallocate y, points to x
    x{};                                      // reallocate x, not pointing to y
} // ^z{}; ^y{}; ^x{}; implicit calls
```

Like C++, construction is implicit on allocation (stack/heap) and destruction is implicit on deallocation. The object and all their fields are constructed/destructed. C∀ also provides new and delete as library routines, which behave like malloc and free, in addition to constructing and destructing objects:

```
{
    ... struct S s = {10}; ...                // allocation, call constructor
}                                             // deallocation, call destructor
struct S * s = new();                         // allocation, call constructor
...
delete( s );                                  // deallocation, call destructor
```

C∀ concurrency uses object lifetime as a means of mutual exclusion and/or synchronization.

2.6 | Parametric Polymorphism

The signature feature of C∀ is parametric-polymorphic routines[10] with routines generalized using a **forall** clause (giving the language its name), which allow separately compiled routines to support generic usage over multiple types. For example, the following sum routine works for any type that supports construction from 0 and addition:

```
forall( otype T | { void ?{}( T *, zero_t ); T ?+?( T, T ); } ) // constraint type, 0 and +
T sum( T a[], size_t size ) {
    T total = { 0 };                          // initialize by 0 constructor
    for ( size_t i = 0; i < size; i += 1 )
        total = total + a[i];                 // select appropriate +
    return total;
}
S sa[5];
int i = sum( sa, 5 );                         // use S's 0 construction and +
```

Type variables can be **otype** or **dtype**. **otype** refers to a *complete type*, *ie*, a type with size, alignment, default constructor, copy constructor, destructor, and assignment operator. **dtype** refers to an *incomplete type*, *eg*, void or a forward-declared type. The builtin types **zero_t** and **one_t** overload constant 0 and 1 for a new types, where both 0 and 1 have special meaning in C.

C∀ provides *traits* to name a group of type assertions, where the trait name allows specifying the same set of assertions in multiple locations, preventing repetition mistakes at each routine declaration:

```
trait sumable( otype T ) {
    void ?{}( T &, zero_t );                  // 0 literal constructor
    T ?+?( T, T );                            // assortment of additions
    T ?+=?( T &, T );
    T ++?( T & );
    T ?++( T & );
};
forall( otype T | sumable( T ) )              // use trait
T sum( T a[], size_t size );
```

Using the return type for overload discrimination, it is possible to write a type-safe alloc based on the C malloc:

```
forall( dtype T | sized(T) ) T * alloc( void ) { return (T *)malloc( sizeof(T) ); }
int * ip = alloc();                           // select type and size from left-hand side
double * dp = alloc();
struct S {...} * sp = alloc();
```

where the return type supplies the type/size of the allocation, which is impossible in most type systems.

3 | CONCURRENCY

At its core, concurrency is based on multiple call-stacks and scheduling threads executing on these stacks. Multiple call stacks (or contexts) and a single thread of execution, called *coroutining*[15,16], does *not* imply concurrency[9, § 2]. In coroutining, the single thread is self-scheduling across the stacks, so execution is deterministic, *ie*, the execution path from input to output is fixed and predictable. A *stackless* coroutine executes on the caller's stack[17] but this approach is restrictive, *eg*, preventing modularization and supporting only iterator/generator-style programming; a *stackful* coroutine executes on its own stack, allowing full generality. Only stackful coroutines are a stepping stone to concurrency.

The transition to concurrency, even for execution with a single thread and multiple stacks, occurs when coroutines also context switch to a *scheduling oracle*, introducing non-determinism from the coroutine perspective[9, § 3]. Therefore, a minimal concurrency system is possible using coroutines (see Section 3.1) in conjunction with a scheduler to decide where to context switch next. The resulting execution system now follows a cooperative threading-model, called *non-preemptive scheduling*.

Because the scheduler is special, it can either be a stackless or stackful coroutine. For stackless, the scheduler performs scheduling on the stack of the current coroutine and switches directly to the next coroutine, so there is one context switch. For stackful, the current coroutine switches to the scheduler, which performs scheduling, and it then switches to the next coroutine, so there are two context switches. A stackful scheduler is often used for simplicity and security.

Regardless of the approach used, a subset of concurrency related challenges start to appear. For the complete set of concurrency challenges to occur, the missing feature is *preemption*, where context switching occurs randomly between any two instructions, often based on a timer interrupt, called *preemptive scheduling*. While a scheduler introduces uncertainty in the order of execution, preemption introduces uncertainty about where context switches occur. Interestingly, uncertainty is necessary for the runtime (operating) system to give the illusion of parallelism on a single processor and increase performance on multiple processors. The reason is that only the runtime has complete knowledge about resources and how to best utilized them. However, the introduction of unrestricted non-determinism results in the need for *mutual exclusion* and *synchronization* to restrict non-determinism for correctness; otherwise, it is impossible to write meaningful programs. Optimal performance in concurrent applications is often obtained by having as much non-determinism as correctness allows.

An important missing feature in C is threading[A]. In modern programming languages, a lack of threading is unacceptable[18,19], and therefore existing and new programming languages must have tools for writing efficient concurrent programs to take advantage of parallelism. As an extension of C, C∀ needs to express these concepts in a way that is as natural as possible to programmers familiar with imperative languages. Furthermore, because C is a system-level language, programmers expect to choose precisely which features they need and which cost they are willing to pay. Hence, concurrent programs should be written using high-level mechanisms, and only step down to lower-level mechanisms when performance bottlenecks are encountered.

3.1 | Coroutines: A Stepping Stone

While the focus of this discussion is concurrency and parallelism, it is important to address coroutines, which are a significant building block of a concurrency system (but not concurrent among themselves). Coroutines are generalized routines allowing execution to be temporarily suspended and later resumed. Hence, unlike a normal routine, a coroutine may not terminate when it returns to its caller, allowing it to be restarted with the values and execution location present at the point of suspension. This capability is accomplished via the coroutine's stack, where suspend/resume context switch among stacks. Because threading design-challenges are present in coroutines, their design effort is relevant, and this effort can be easily exposed to programmers giving them a useful new programming paradigm because a coroutine handles the class of problems that need to retain state between calls, *eg*, plugins, device drivers, and finite-state machines. Therefore, the two fundamental features of the core C∀ coroutine-API are independent call-stacks and `suspend`/`resume` operations.

For example, a problem made easier with coroutines is unbounded generators, *eg*, generating an infinite sequence of Fibonacci numbers

$$\mathsf{fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathsf{fib}(n-1) + \mathsf{fib}(n-2) & n \ge 2 \end{cases}$$

[A] While the C11 standard defines a `threads.h` header, it is minimal and defined as optional. As such, library support for threading is far from widespread. At the time of writing the paper, neither `gcc` nor `clang` support `threads.h` in their standard libraries.

```
int f1, f2, state = 1;   // single global variables
int fib() {
    int fn;
    switch ( state ) {  // explicit execution state
      case 1: fn = 0;  f1 = fn;  state = 2;  break;
      case 2: fn = 1;  f2 = f1;  f1 = fn;  state = 3;  break;
      case 3: fn = f1 + f2;  f2 = f1;  f1 = fn;  break;
    }
    return fn;
}
int main() {

    for ( int i = 0; i < 10; i += 1 ) {
        printf( "%d\n", fib() );
    }
}
```

(A) 3 States: global variables

```
#define FIB_INIT { 0, 1 }
typedef struct { int f2, f1; } Fib;
int fib( Fib * f ) {

    int ret = f->f2;
    int fn = f->f1 + f->f2;
    f->f2 = f->f1; f->f1 = fn;

    return ret;
}
int main() {
    Fib f1 = FIB_INIT, f2 = FIB_INIT;
    for ( int i = 0; i < 10; i += 1 ) {
        printf( "%d %d\n", fib( &f1 ), fib( &f2 ) );
    }
}
```

(B) 1 State: external variables

FIGURE 1 C Fibonacci Implementations

```
coroutine Fib { int fn; };
void main( Fib & fib ) with( fib ) {
    int f1, f2;
    fn = 0;  f1 = fn;  suspend();
    fn = 1;  f2 = f1;  f1 = fn;  suspend();
    for ( ;; ) {
        fn = f1 + f2;  f2 = f1;  f1 = fn;  suspend();
    }
}
int next( Fib & fib ) with( fib ) {
    resume( fib );
    return fn;
}
int main() {
    Fib f1, f2;
    for ( int i = 1; i <= 10; i += 1 ) {
        sout | next( f1 ) | next( f2 );
    }
}
```

(A) 3 States, internal variables

```
coroutine Fib { int ret; };
void main( Fib & f ) with( fib ) {
    int fn, f1 = 1, f2 = 0;
    for ( ;; ) {
        ret = f2;

        fn = f1 + f2;  f2 = f1;  f1 = fn; suspend();
    }
}
int next( Fib & fib ) with( fib ) {
    resume( fib );
    return ret;
}
```

(B) 1 State, internal variables

FIGURE 2 C∀ Coroutine Fibonacci Implementations

where Figure 1 shows conventional approaches for writing a Fibonacci generator in C. Figure 1 (A) illustrates the following problems: unique unencapsulated global variables necessary to retain state between calls, only one Fibonacci generator, and execution state must be explicitly retained via explicit state variables. Figure 1 (B) addresses these issues: unencapsulated program global variables become encapsulated structure variables, unique global variables are replaced by multiple Fibonacci objects, and explicit execution state is removed by precomputing the first two Fibonacci numbers and returning fib($n-2$).

Using a coroutine, it is possible to express the Fibonacci formula directly without any of the C problems. Figure 2 (A) creates a **coroutine** type, **coroutine** Fib { **int** fn; }, which provides communication, fn, for the *coroutine main*, main, which runs on the coroutine stack, and possibly multiple interface routines, *eg*, next. Like the structure in Figure 1 (B), the coroutine type allows multiple instances, where instances of this type are passed to the (overloaded) coroutine main. The coroutine main's stack holds the state for the next generation, f1 and f2, and the code represents the three states in the Fibonacci formula via the three suspend points, to context switch back to the caller's resume. The interface routine next, takes a Fibonacci instance and context

```
coroutine Format {
    char ch;   // used for communication
    int g, b;  // global because used in destructor
};
void main( Format & fmt ) with( fmt ) {
    for ( ;; ) {
        for ( g = 0; g < 5; g += 1 ) {      // group
            for ( b = 0; b < 4; b += 1 ) { // block
                suspend();
                sout | ch;              // separator
            }
            sout | "  ";               // separator
        }
        sout | nl;
    }
}
void ?{}( Format & fmt ) { resume( fmt ); }
void ^?{}( Format & fmt ) with( fmt ) {
    if ( g != 0 || b != 0 ) sout | nl;
}
void format( Format & fmt ) {
    resume( fmt );
}
int main() {
    Format fmt;
    eof: for ( ;; ) {
        sin | fmt.ch;
      if ( eof( sin ) ) break eof;
        format( fmt );
    }
}
```

(A) C∀ Coroutine

```
struct Format {
    char ch;
    int g, b;
};
void format( struct Format * fmt ) {
    if ( fmt->ch != -1 ) {      // not EOF ?
        printf( "%c", fmt->ch );
        fmt->b += 1;
        if ( fmt->b == 4 ) {  // block
            printf( "  " );      // separator
            fmt->b = 0;
            fmt->g += 1;
        }
        if ( fmt->g == 5 ) {  // group
            printf( "\n" );     // separator
            fmt->g = 0;
        }
    } else {
        if ( fmt->g != 0 || fmt->b != 0 ) printf( "\n" );
    }
}
int main() {
    struct Format fmt = { 0, 0, 0 };
    for ( ;; ) {
        scanf( "%c", &fmt.ch );
      if ( feof( stdin ) ) break;
        format( &fmt );
    }
    fmt.ch = -1;
    format( &fmt );
}
```

(B) C Linearized

FIGURE 3 Formatting text into lines of 5 blocks of 4 characters.

switches to it using resume; on restart, the Fibonacci field, fn, contains the next value in the sequence, which is returned. The first resume is special because it allocates the coroutine stack and cocalls its coroutine main on that stack; when the coroutine main returns, its stack is deallocated. Hence, Fib is an object at creation, transitions to a coroutine on its first resume, and transitions back to an object when the coroutine main finishes. Figure 2 (B) shows the coroutine version of the C version in Figure 1 (B). Coroutine generators are called *output coroutines* because values are only returned.

Figure 3 (A) shows an *input coroutine*, Format, for restructuring text into groups of characters of fixed-size blocks. For example, the input of the left is reformatted into the output on the right.

input	**output**

`abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz`	`abcd  efgh  ijkl  mnop  qrst`
	`uvwx  yzab  cdef  ghij  klmn`
	`opqr  stuv  wxyz`

The example takes advantage of resuming a coroutine in the constructor to prime the loops so the first character sent for formatting appears inside the nested loops. The destructor provides a newline, if formatted text ends with a full line. Figure 3 (B) shows the C equivalent formatter, where the loops of the coroutine are flattened (linearized) and rechecked on each call because execution location is not retained between calls. (Linearized code is the bane of device drivers.)

The previous examples are *asymmetric (semi) coroutine*s because one coroutine always calls a resuming routine for another coroutine, and the resumed coroutine always suspends back to its last resumer, similar to call/return for normal routines. However,

```
coroutine Prod {
    Cons & c;
    int N, money, receipt;
};
void main( Prod & prod ) with( prod ) {
    // 1st resume starts here
    for ( int i = 0; i < N; i += 1 ) {
        int p1 = random( 100 ), p2 = random( 100 );
        sout | p1 | "  " | p2;
        int status = delivery( c, p1, p2 );
        sout | " $" | money | nl | status;
        receipt += 1;
    }
    stop( c );
    sout | "prod stops";
}
int payment( Prod & prod, int money ) {
    prod.money = money;
    resume( prod );
    return prod.receipt;
}
void start( Prod & prod, int N, Cons &c ) {
    &prod.c = &c;
    prod.[N, receipt] = [N, 0];
    resume( prod );
}
int main() {
    Prod prod;
    Cons cons = { prod };
    start( prod, 5, cons );
}
```

```
coroutine Cons {
    Prod & p;
    int p1, p2, status;
    _Bool done;
};
void ?{}( Cons & cons, Prod & p ) {
    &cons.p = &p;
    cons.[status, done ] = [0, false];
}
void ^?{}( Cons & cons ) {}
void main( Cons & cons ) with( cons ) {
    // 1st resume starts here
    int money = 1, receipt;
    for ( ; ! done; ) {
        sout | p1 | "  " | p2 | nl | " $" | money;
        status += 1;
        receipt = payment( p, money );
        sout | " #" | receipt;
        money += 1;
    }
    sout | "cons stops";
}
int delivery( Cons & cons, int p1, int p2 ) {
    cons.[p1, p2] = [p1, p2];
    resume( cons );
    return cons.status;
}
void stop( Cons & cons ) {
    cons.done = true;
    resume( cons );
}
```

FIGURE 4 Producer / consumer: resume-resume cycle, bi-directional communication

resume and suspend context switch among existing stack-frames, rather than create new ones so there is no stack growth. *Symmetric (full) coroutine*s have a coroutine call to a resuming routine for another coroutine, and its coroutine main calls another resuming routine, which eventually forms a resuming-call cycle. (The trivial cycle is a coroutine resuming itself.) This control flow is similar to recursion for normal routines, but again there is no stack growth from the context switch.

Figure 4 shows a producer/consumer symmetric-coroutine performing bi-directional communication. Since the solution involves a full-coroutining cycle, the program main creates one coroutine in isolation, passes this coroutine to its partner, and closes the cycle at the call to start. The start routine communicates both the number of elements to be produced and the consumer into the producer's coroutine-structure. Then the resume to prod creates prod's stack with a frame for prod's coroutine main at the top, and context switches to it. prod's coroutine main starts, creates local variables that are retained between coroutine activations, and executes N iterations, each generating two random values, calling the consumer to deliver the values, and printing the status returned from the consumer.

The producer call to delivery transfers values into the consumer's communication variables, resumes the consumer, and returns the consumer status. For the first resume, cons's stack is initialized, creating local variables retained between subsequent activations of the coroutine. The consumer iterates until the done flag is set, prints the values delivered by the producer, increments status, and calls back to the producer via payment, and on return from payment, prints the receipt from the producer and increments money (inflation). The call from the consumer to payment introduces the cycle between producer and consumer. When payment is called, the consumer copies values into the producer's communication variable and a resume is executed. The context switch restarts the producer at the point where it last context switched, so it continues in delivery after the resume.

delivery returns the status value in prod's coroutine main, where the status is printed. The loop then repeats calling delivery, where each call resumes the consumer coroutine. The context switch to the consumer continues in payment. The consumer increments and returns the receipt to the call in cons's coroutine main. The loop then repeats calling payment, where each call resumes the producer coroutine.

After iterating N times, the producer calls stop. The done flag is set to stop the consumer's execution and a resume is executed. The context switch restarts cons in payment and it returns with the last receipt. The consumer terminates its loops because done is true, its main terminates, so cons transitions from a coroutine back to an object, and prod reactivates after the resume in stop. stop returns and prod's coroutine main terminates. The program main restarts after the resume in start. start returns and the program main terminates.

3.2 | Coroutine Implementation

A significant implementation challenge for coroutines (and threads, see Section 3.3) is adding extra fields and executing code after/before the coroutine constructor/destructor and coroutine main to create/initialize/de-initialize/destroy extra fields and the stack. There are several solutions to this problem and the chosen option forced the C∀ coroutine design.

Object-oriented inheritance provides extra fields and code in a restricted context, but it requires programmers to explicitly perform the inheritance:

```
class mycoroutine inherits baseCoroutine { ... }
```

and the programming language (and possibly its tool set, *eg*, debugger) may need to understand baseCoroutine because of the stack. Furthermore, the execution of constructors/destructors is in the wrong order for certain operations. For example, for threads if the thread is implicitly started, it must start *after* all constructors, because the thread relies on a completely initialized object, but the inherited constructor runs *before* the derived.

An alternative is composition:

```
struct mycoroutine {
    ... // declarations
    baseCoroutine dummy; // composition, last declaration
}
```

which also requires an explicit declaration that must be the last one to ensure correct initialization order. However, there is nothing preventing wrong placement or multiple declarations.

For coroutines as for threads, many implementations are based on routine pointers or routine objects[20,21,22,23]. For example, Boost implements coroutines in terms of four functor object-types:

```
asymmetric_coroutine<>::pull_type
asymmetric_coroutine<>::push_type
symmetric_coroutine<>::call_type
symmetric_coroutine<>::yield_type
```

Similarly, the canonical threading paradigm is often based on routine pointers, *eg*, pthreads[20], C♯[24], Go[6], and Scala[25]. However, the generic thread-handle (identifier) is limited (few operations), unless it is wrapped in a custom type.

```
void mycor( coroutine_t cid, void * arg ) {
    int * value = (int *)arg;                    // type unsafe, pointer-size only
    // Coroutine body
}
int main() {
    int input = 0, output;
    coroutine_t cid = coroutine_create( &mycor, (void *)&input ); // type unsafe, pointer-size only
    coroutine_resume( cid, (void *)input, (void **)&output ); // type unsafe, pointer-size only
}
```

Since the custom type is simple to write in C∀ and solves several issues, added support for routine/lambda-based coroutines adds very little.

Note, the type coroutine_t must be an abstract handle to the coroutine, because the coroutine descriptor and its stack are non-copyable. Copying the coroutine descriptor results in copies being out of date with the current state of the stack. Correspondingly, copying the stack results is copies being out of date with the coroutine descriptor, and pointers in the stack being out of date to data on the stack. (There is no mechanism in C to find all stack-specific pointers and update them as part of a copy.)

The selected approach is to use language support by introducing a new kind of aggregate (structure):

```
coroutine Fibonacci {
    int fn; // communication variables
};
```

The **coroutine** keyword means the compiler (and tool set) can find and inject code where needed. The downside of this approach is that it makes coroutine a special case in the language. Users wanting to extend coroutines or build their own for various reasons can only do so in ways offered by the language. Furthermore, implementing coroutines without language supports also displays the power of a programming language. While this is ultimately the option used for idiomatic C∀ code, coroutines and threads can still be constructed without language support. The reserved keyword simply eases use for the common case.

Part of the mechanism to generalize coroutines is using a C∀ trait, which defines a coroutine as anything satisfying the trait is_coroutine, and this trait restricts the available set of coroutine-manipulation routines:

```
trait is_coroutine( dtype T ) {
    void main( T & );
    coroutine_desc * get_coroutine( T & );
};
forall( dtype T | is_coroutine(T) ) void suspend( T & );
forall( dtype T | is_coroutine(T) ) void resume( T & );
```

The **dtype** property provides no implicit copying operations and the is_coroutine trait provides no explicit copying operations, so all coroutines must be passed by reference (pointer). The routine definitions ensures there is a statically-typed main routine that is the starting point (first stack frame) of a coroutine, and a mechanism to get (read) the currently executing coroutine handle. The main routine has no return value or additional parameters because the coroutine type allows an arbitrary number of interface routines with corresponding arbitrary typed input/output values versus fixed ones. The advantage of this approach is that users can easily create different types of coroutines, *eg*, changing the memory layout of a coroutine is trivial when implementing the get_coroutine routine, and possibly redefining suspend and resume. The C∀ keyword **coroutine** implicitly implements the getter and forward declarations required for implementing the coroutine main:

```
coroutine MyCor {          struct MyCor {           static inline coroutine_desc *      void main( MyCor * this );
    int value;        ⇒        int value;           get_coroutine( MyCor & this ) {
                               coroutine_desc cor;      return &this.cor;
};                         };                       }
```

The combination of these two approaches allows an easy and concise specification to coroutining (and concurrency) for normal users, while more advanced users have tighter control on memory layout and initialization.

3.3 | Thread Interface

Both user and kernel threads are supported, where user threads provide concurrency and kernel threads provide parallelism. Like coroutines and for the same design reasons, the selected approach for user threads is to use language support by introducing a new kind of aggregate (structure) and a C∀ trait:

```
thread myThread {                     trait is_thread( dtype T ) {
    // communication variables            void main( T & );
};                                        thread_desc * get_thread( T & );
                                          void ^?{}( T & mutex );
                                      };
```

(The qualifier **mutex** for the destructor parameter is discussed in Section 5.) Like a coroutine, the statically-typed main routine is the starting point (first stack frame) of a user thread. The difference is that a coroutine borrows a thread from its caller, so the first thread resuming a coroutine creates an instance of main; whereas, a user thread receives its own thread from the runtime system, which starts in main as some point after the thread constructor is run.[B] No return value or additional parameters are

[B]The main routine is already a special routine in C, *ie*, where the program's initial thread begins, so it is a natural extension of this semantics to use overloading to declare mains for user coroutines and threads.

necessary for this routine because the task type allows an arbitrary number of interface routines with corresponding arbitrary typed input/output values.

For user threads to be useful, it must be possible to start and stop the underlying thread, and wait for it to complete execution. While using an API such as fork and join is relatively common, such an interface is awkward and unnecessary. A simple approach is to use allocation/deallocation principles, and have threads implicitly fork after construction and join before destruction.

```
thread World {};
void main( World & this ) {
    sout | "World!";
}
int main() {
    World w[10];                              // implicit forks after creation
    sout | "Hello ";                          // "Hello " and 10 "World!" printed concurrently
}                                             // implicit joins before destruction
```

This semantics ensures a thread is started and stopped exactly once, eliminating some programming error, and scales to multiple threads for basic (termination) synchronization. This tree-structure (lattice) create/delete from C block-structure is generalized by using dynamic allocation, so threads can outlive the scope in which they are created, much like dynamically allocating memory lets objects outlive the scope in which they are created.

```
int main() {
    MyThread * heapLive;
    {
        MyThread blockLive;                   // fork block-based thread
        heapLive = new( MyThread );           // fork heap-based thread
        ...
    }                                         // join block-based thread
    ...
    delete( heapLive );                       // join heap-based thread
}
```

The heap-based approach allows arbitrary thread-creation topologies, with respect to fork/join-style concurrency.

Figure 5 shows concurrently adding the rows of a matrix and then totalling the subtotals sequentially, after all the row threads have terminated. The program uses heap-based threads because each thread needs different constructor values. (Python provides a simple iteration mechanism to initialize array elements to different values allowing stack allocation.) The allocation/deallocation pattern appears unusual because allocated objects are immediately deallocated without any intervening code. However, for threads, the deletion provides implicit synchronization, which is the intervening code. While the subtotals are added in linear order rather than completion order, which slightly inhibits concurrency, the computation is restricted by the critical-path thread (*ie*, the thread that takes the longest), and so any inhibited concurrency is very small as totalling the subtotals is trivial.

4 | MUTUAL EXCLUSION / SYNCHRONIZATION

Uncontrolled non-deterministic execution is meaningless. To reestablish meaningful execution requires mechanisms to reintroduce determinism, *ie*, restrict non-determinism, called mutual exclusion and synchronization, where mutual exclusion is an access-control mechanism on data shared by threads, and synchronization is a timing relationship among threads[9, § 4]. Since many deterministic challenges appear with the use of mutable shared state, some languages/libraries disallow it, *eg*, Erlang[7], Haskell[26], Akka[27] (Scala). In these paradigms, interaction among concurrent objects is performed by stateless message-passing[28,29,30] or other paradigms closely related to networking concepts, *eg*, channels[5,6]. However, in call/return-based languages, these approaches force a clear distinction, *ie*, introduce a new programming paradigm between regular and concurrent computation, *eg*, routine call versus message passing. Hence, a programmer must learn and manipulate two sets of design patterns. While this distinction can be hidden away in library code, effective use of the library still has to take both paradigms into account. In contrast, approaches based on stateful models more closely resemble the standard call/return programming-model, resulting in a single programming paradigm.

At the lowest level, concurrent control is implemented by atomic operations, upon which different kinds of locking mechanisms are constructed, *eg*, semaphores[31], barriers, and path expressions[32]. However, for productivity it is always desirable to

```
thread Adder { int * row, cols, & subtotal; }                    // communication variables
void ?{}( Adder & adder, int row[], int cols, int & subtotal ) {
    adder.[ row, cols, &subtotal ] = [ row, cols, &subtotal ];
}
void main( Adder & adder ) with( adder ) {
    subtotal = 0;
    for ( int c = 0; c < cols; c += 1 ) { subtotal += row[c]; }
}
int main() {
    const int rows = 10, cols = 1000;
    int matrix[rows][cols], subtotals[rows], total = 0;
    // read matrix
    Adder * adders[rows];
    for ( int r = 0; r < rows; r += 1 ) {                        // start threads to sum rows
        adders[r] = new( matrix[r], cols, &subtotals[r] );
    }
    for ( int r = 0; r < rows; r += 1 ) {                        // wait for threads to finish
        delete( adders[r] );                                     // termination join
        total += subtotals[r];                                   // total subtotal
    }
    sout | total;
}
```

FIGURE 5 Concurrent Matrix Summation

use the highest-level construct that provides the necessary efficiency[1]. A newer approach for restricting non-determinism is transactional memory[33]. While this approach is pursued in hardware[34] and system languages, like C++[35], the performance and feature set is still too restrictive to be the main concurrency paradigm for system languages, which is why it is rejected as the core paradigm for concurrency in C∀.

One of the most natural, elegant, and efficient mechanisms for mutual exclusion and synchronization for shared-memory systems is the *monitor*. First proposed by Brinch Hansen[36] and later described and extended by C.A.R. Hoare[37], many concurrent programming-languages provide monitors as an explicit language construct: *eg*, Concurrent Pascal[38], Mesa[39], Modula[40], Turing[41], Modula-3[42], NeWS[43], Emerald[44], μC++[45] and Java[4]. In addition, operating-system kernels and device drivers have a monitor-like structure, although they often use lower-level primitives such as mutex locks or semaphores to simulate monitors. For these reasons, C∀ selected monitors as the core high-level concurrency-construct, upon which higher-level approaches can be easily constructed.

4.1 | Mutual Exclusion

A group of instructions manipulating a specific instance of shared data that must be performed atomically is called an (individual) *critical-section*[46]. The generalization is called a *group critical-section*[47], where multiple tasks with the same session may use the resource simultaneously, but different sessions may not use the resource simultaneously. The readers/writer problem[48] is an instance of a group critical-section, where readers have the same session and all writers have a unique session. *Mutual exclusion* enforces that the correct kind and number of threads are using a critical section.

However, many solutions exist for mutual exclusion, which vary in terms of performance, flexibility and ease of use. Methods range from low-level locks, which are fast and flexible but require significant attention for correctness, to higher-level concurrency techniques, which sacrifice some performance to improve ease of use. Ease of use comes by either guaranteeing some problems cannot occur, *eg*, deadlock free, or by offering a more explicit coupling between shared data and critical section. For example, the C++ std::atomic<T> offers an easy way to express mutual-exclusion on a restricted set of operations, *eg*, reading/writing, for numerical types. However, a significant challenge with locks is composability because it takes careful organization for multiple locks to be used while preventing deadlock. Easing composability is another feature higher-level mutual-exclusion mechanisms can offer.

4.2 | Synchronization

Synchronization enforces relative ordering of execution, and synchronization tools provide numerous mechanisms to establish these timing relationships. Low-level synchronization primitives offer good performance and flexibility at the cost of ease of use; higher-level mechanisms often simplify usage by adding better coupling between synchronization and data, *eg*, message passing, or offering a simpler solution to otherwise involved challenges, *eg*, barrier lock. Often synchronization is used to order access to a critical section, *eg*, ensuring a reader thread is the next kind of thread to enter a critical section. If a writer thread is scheduled for next access, but another reader thread acquires the critical section first, that reader *barged*. Barging can result in staleness/freshness problems, where a reader barges ahead of a writer and reads temporally stale data, or a writer barges ahead of another writer overwriting data with a fresh value preventing the previous value from ever being read (lost computation). Preventing or detecting barging is an involved challenge with low-level locks, which can be made much easier by higher-level constructs. This challenge is often split into two different approaches: barging avoidance and barging prevention. Algorithms that allow a barger, but divert it until later using current synchronization state (flags), are avoiding the barger; algorithms that preclude a barger from entering during synchronization in the critical section prevent barging completely. Techniques like baton-passing locks[49] between threads instead of unconditionally releasing locks is an example of barging prevention.

5 | MONITOR

A **monitor** is a set of routines that ensure mutual exclusion when accessing shared state. More precisely, a monitor is a programming technique that binds mutual exclusion to routine scope, as opposed to locks, where mutual-exclusion is defined by acquire/release calls, independent of lexical context (analogous to block and heap storage allocation). The strong association with the call/return paradigm eases programmability, readability and maintainability, at a slight cost in flexibility and efficiency.

Note, like coroutines/threads, both locks and monitors require an abstract handle to reference them, because at their core, both mechanisms are manipulating non-copyable shared-state. Copying a lock is insecure because it is possible to copy an open lock and then use the open copy when the original lock is closed to simultaneously access the shared data. Copying a monitor is secure because both the lock and shared data are copies, but copying the shared data is meaningless because it no longer represents a unique entity. As for coroutines/tasks, the **dtype** property provides no implicit copying operations and the is_monitor trait provides no explicit copying operations, so all locks/monitors must be passed by reference (pointer).

```
trait is_monitor( dtype T ) {
    monitor_desc * get_monitor( T & );
    void ^?{}( T & mutex );
};
```

5.1 | Mutex Acquisition

While correctness implies a monitor's mutual exclusion is acquired and released, there are implementation options about when and where the locking/unlocking occurs. (Much of this discussion also applies to basic locks.) For example, a monitor may need to be passed through multiple helper routines before it becomes necessary to acquire the monitor mutual-exclusion.

```
monitor Aint { int cnt; };                          // atomic integer counter
void ?{}( Aint & nomutex this ) with( this ) { cnt = 0; } // constructor
int ?=?( Aint & mutex_opt lhs, int rhs ) with( lhs ) { cnt = rhs; } // conversions
void ?{}( int & this, Aint & mutex_opt v ) { this = v.cnt; }
int ?=?( int & lhs, Aint & mutex_opt rhs ) with( rhs ) { lhs = cnt; }
int ++?( Aint & mutex_opt this ) with( this ) { return ++cnt; } // increment
```

The Aint constructor, ?{}, uses the **nomutex** qualifier indicating mutual exclusion is unnecessary during construction because an object is inaccessible (private) until after it is initialized. (While a constructor may publish its address into a global variable, doing so generates a race-condition.) The conversion operators for initializing and assigning with a normal integer only need **mutex**, if reading/writing the implementation type is not atomic. Finally, the prefix increment operato, ++?, is normally **mutex** to protect the incrementing from race conditions, unless there is an atomic increment instruction for the implementation type.

The atomic counter is used without any explicit mutual-exclusion and provides thread-safe semantics, which is similar to the C++ template std::atomic.

```
Aint x, y, z;
++x; ++y; ++z;                                  // safe increment by multiple threads
x = 2; y = 2; z = 2;                            // conversions
int i = x, j = y, k = z;
i = x; j = y; k = z;
```

For maximum usability, monitors have *multi-acquire* semantics allowing a thread to acquire it multiple times without deadlock.

```
monitor M { ... } m;
void foo( M & mutex m ) { ... }                 // acquire mutual exclusion
void bar( M & mutex m ) {                       // acquire mutual exclusion
    ... foo( m ); ...                           // reacquire mutual exclusion
}
bar( m );                                       // nested monitor call
```

The benefit of mandatory monitor qualifiers is self-documentation, but requiring both **mutex** and **nomutex** for all monitor parameters is redundant. Instead, the semantics have one qualifier as the default, and the other required. For example, make the safe **mutex** qualifier the default because assuming **nomutex** may cause subtle errors. Alternatively, make the unsafe **nomutex** qualifier the default because it is the *normal* parameter semantics while **mutex** parameters are rare. Providing a default qualifier implies knowing whether a parameter is a monitor. Since C∀ relies heavily on traits as an abstraction mechanism, the distinction between a type that is a monitor and a type that looks like a monitor can become blurred. For this reason, C∀ requires programmers to identify the kind of parameter with the **mutex** keyword and uses no keyword to mean **nomutex**.

The next semantic decision is establishing which parameter *types* may be qualified with **mutex**. Given:

```
monitor M { ... }
int f1( M & mutex m );
int f2( M * mutex m );
int f3( M * mutex m[] );
int f4( stack( M * ) & mutex m );
```

the issue is that some of these parameter types are composed of multiple objects. For f1, there is only a single parameter object. Adding indirection in f2 still identifies a single object. However, the matrix in f3 introduces multiple objects. While shown shortly, multiple acquisition is possible; however array lengths are often unknown in C. This issue is exacerbated in f4, where the data structure must be safely traversed to acquire all of its elements.

To make the issue tractable, C∀ only acquires one monitor per parameter with at most one level of indirection. However, there is an ambiguity in the C type-system with respects to arrays. Is the argument for f2 a single object or an array of objects? If it is an array, only the first element of the array is acquired, which seems unsafe; hence, **mutex** is disallowed for array parameters.

```
int f1( M & mutex m );                          // allowed: recommended case
int f2( M * mutex m );                          // disallowed: could be an array
int f3( M mutex m[ ] );                         // disallowed: array length unknown
int f4( M ** mutex m );                         // disallowed: could be an array
int f5( M * mutex m[ ] );                       // disallowed: array length unknown
```

For object-oriented monitors, calling a mutex member *implicitly* acquires mutual exclusion of the receiver object, rec.foo(...). C∀ has no receiver, and hence, must use an explicit mechanism to specify which object acquires mutual exclusion. A positive consequence of this design decision is the ability to support multi-monitor routines.

```
int f( M & mutex x, M & mutex y );              // multiple monitor parameter of any type
M m1, m2;
f( m1, m2 );
```

(While object-oriented monitors can be extended with a mutex qualifier for multiple-monitor members, no prior example of this feature could be found.) In practice, writing multi-locking routines that do not deadlock is tricky. Having language support for such a feature is therefore a significant asset for C∀.

The capability to acquire multiple locks before entering a critical section is called *bulk acquire* (see Section 9 for implementation details). In the previous example, C∀ guarantees the order of acquisition is consistent across calls to different routines using the same monitors as arguments. This consistent ordering means acquiring multiple monitors is safe from deadlock. However, users can force the acquiring order. For example, notice the use of **mutex/nomutex** and how this affects the acquiring order:

```
void foo( M & mutex m1, M & mutex m2 );              // acquire m1 and m2
void bar( M & mutex m1, M & /* nomutex */ m2 ) {  // acquire m1
    ... foo( m1, m2 ); ...                             // acquire m2
}
void baz( M & /* nomutex */ m1, M & mutex m2 ) {  // acquire m2
    ... foo( m1, m2 ); ...                             // acquire m1
}
```

The multi-acquire semantics allows bar or baz to acquire a monitor lock and reacquire it in foo. In the calls to bar and baz, the monitors are acquired in opposite order.

However, such use leads to lock acquiring order problems resulting in deadlock[50], where detecting it requires dynamic tracking of monitor calls, and dealing with it requires rollback semantics[51]. In C∀, a safety aid is provided by using bulk acquire of all monitors to shared objects, whereas other monitor systems provide no aid. While C∀ provides only a partial solution, it handles many useful cases, *eg*:

```
monitor BankAccount { ... };
void deposit( BankAccount & mutex b, int deposit );
void transfer( BankAccount & mutex my, BankAccount & mutex your, int me2you ) {
    deposit( my, -me2you );                              // debit
    deposit( your, me2you );                             // credit
}
```

This example shows a trivial solution to the bank-account transfer problem. Without multi- and bulk acquire, the solution to this problem requires careful engineering.

5.2 | mutex statement

The monitor call-semantics associate all locking semantics to routines. Like Java, C∀ offers an alternative **mutex** statement to reduce refactoring and naming.

```
monitor M { ... };
void foo( M & mutex m1, M & mutex m2 ) {
    // critical section
}
void bar( M & m1, M & m2 ) {
    foo( m1, m2 );
}
```

routine call

```
void bar( M & m1, M & m2 ) {
    mutex( m1, m2 ) {   // remove refactoring and naming
        // critical section
    }
}
```

mutex statement

6 | SCHEDULING

While monitor mutual-exclusion provides safe access to shared data, the monitor data may indicate that a thread accessing it cannot proceed. For example, Figure 6 shows a bounded buffer that may be full/empty so produce/consumer threads must block. Leaving the monitor and trying again (busy waiting) is impractical for high-level programming. Monitors eliminate busy waiting by providing synchronization to schedule threads needing access to the shared data, where threads block versus spinning. Synchronization is generally achieved with internal[37] or external[52, § 2.9.2] scheduling, where *scheduling* defines which thread acquires the critical section next. *Internal scheduling* is characterized by each thread entering the monitor and making an individual decision about proceeding or blocking, while *external scheduling* is characterized by an entering thread making a decision about proceeding for itself and on behalf of other threads attempting entry.

Figure 6 (A) shows a C∀ generic bounded-buffer with internal scheduling, where producers/consumers enter the monitor, see the buffer is full/empty, and block on an appropriate condition lock, full/empty. The wait routine atomically blocks the calling thread and implicitly releases the monitor lock(s) for all monitors in the routine's parameter list. The appropriate condition lock is signalled to unblock an opposite kind of thread after an element is inserted/removed from the buffer. Signalling is unconditional, because signalling an empty condition lock does nothing.

Signalling semantics cannot have the signaller and signalled thread in the monitor simultaneously, which means:

```
forall( otype T ) { // distribute forall
    monitor Buffer {
        condition full, empty;
        int front, back, count;
        T elements[10];
    };
    void ?{}( Buffer(T) & buffer ) with(buffer) {
        [front, back, count] = 0;
    }

    void insert( Buffer(T) & mutex buffer, T elem )
                with(buffer) {
        if ( count == 10 ) wait( empty );
        // insert elem into buffer
        signal( full );
    }
    T remove( Buffer(T) & mutex buffer ) with(buffer) {
        if ( count == 0 ) wait( full );
        // remove elem from buffer
        signal( empty );
        return elem;
    }
}
```

(A) Internal Scheduling

```
forall( otype T ) { // distribute forall
    monitor Buffer {

        int front, back, count;
        T elements[10];
    };
    void ?{}( Buffer(T) & buffer ) with(buffer) {
        [front, back, count] = 0;
    }
    T remove( Buffer(T) & mutex buffer ); // forward
    void insert( Buffer(T) & mutex buffer, T elem )
                with(buffer) {
        if ( count == 10 ) waitfor( remove, buffer );
        // insert elem into buffer

    }
    T remove( Buffer(T) & mutex buffer ) with(buffer) {
        if ( count == 0 ) waitfor( insert, buffer );
        // remove elem from buffer

        return elem;
    }
}
```

(B) External Scheduling

FIGURE 6 Generic Bounded-Buffer

1. The signalling thread returns immediately, and the signalled thread continues.
2. The signalling thread continues and the signalled thread is marked for urgent unblocking at the next scheduling point (exit/wait).
3. The signalling thread blocks but is marked for urgrent unblocking at the next scheduling point and the signalled thread continues.

The first approach is too restrictive, as it precludes solving a reasonable class of problems, *eg*, dating service (see Figure 7). C∀ supports the next two semantics as both are useful. Finally, while it is common to store a condition as a field of the monitor, in C∀, a condition variable can be created/stored independently. Furthermore, a condition variable is tied to a *group* of monitors on first use, called *branding*, which means that using internal scheduling with distinct sets of monitors requires one condition variable per set of monitors.

Figure 6 (B) shows a C∀ generic bounded-buffer with external scheduling, where producers/consumers detecting a full/empty buffer block and prevent more producers/consumers from entering the monitor until there is a free/empty slot in the buffer. External scheduling is controlled by the **waitfor** statement, which atomically blocks the calling thread, releases the monitor lock, and restricts the routine calls that can next acquire mutual exclusion. If the buffer is full, only calls to remove can acquire the buffer, and if the buffer is empty, only calls to insert can acquire the buffer. Threads making calls to routines that are currently excluded, block outside of (external to) the monitor on a calling queue, versus blocking on condition queues inside of (internal to) the monitor. External scheduling allows users to wait for events from other threads without concern of unrelated events occurring. The mechnaism can be done in terms of control flow, *eg*, Ada accept or μC++ _Accept, or in terms of data, *eg*, Go channels. While both mechanisms have strengths and weaknesses, this project uses a control-flow mechanism to stay consistent with other language semantics. Two challenges specific to C∀ for external scheduling are loose object-definitions (see Section 6.2) and multiple-monitor routines (see Section 6.3).

For internal scheduling, non-blocking signalling (as in the producer/consumer example) is used when the signaller is providing the cooperation for a waiting thread; the signaller enters the monitor and changes state, detects a waiting threads that can use the state, performs a non-blocking signal on the condition queue for the waiting thread, and exits the monitor to run concurrently.

```
enum { CCodes = 20 };
monitor DS {
    int GirlPhNo, BoyPhNo;
    condition Girls[CCodes], Boys[CCodes];
    condition exchange;
};
int girl( DS & mutex ds, int phNo, int ccode ) {
    if ( is_empty( Boys[ccode] ) ) {
        wait( Girls[ccode] );
        GirlPhNo = phNo;
        signal( exchange );
    } else {
        GirlPhNo = phNo;
        signal( Boys[ccode] );
        wait( exchange );
    } // if
    return BoyPhNo;
}
int boy( DS & mutex ds, int phNo, int ccode ) {
    // as above with boy/girl interchanged
}
```

(A) signal

```
monitor DS {
    int GirlPhNo, BoyPhNo;
    condition Girls[CCodes], Boys[CCodes];

};
int girl( DS & mutex ds, int phNo, int ccode ) {
    if ( is_empty( Boys[ccode] ) ) { // no compatible
        wait( Girls[ccode] ); // wait for boy
        GirlPhNo = phNo; // make phone number available

    } else {
        GirlPhNo = phNo; // make phone number available
        signal_block( Boys[ccode] ); // restart boy

    } // if
    return BoyPhNo;
}
int boy( DS & mutex ds, int phNo, int ccode ) {
    // as above with boy/girl interchanged
}
```

(B) signal_block

FIGURE 7 Dating service.

The waiter unblocks next from the urgent queue, uses/takes the state, and exits the monitor. Blocking signalling is the reverse, where the waiter is providing the cooperation for the signalling thread; the signaller enters the monitor, detects a waiting thread providing the necessary state, performs a blocking signal to place it on the urgent queue and unblock the waiter. The waiter changes state and exits the monitor, and the signaller unblocks next from the urgent queue to use/take the state.

Figure 7 shows a dating service demonstrating non-blocking and blocking signalling. The dating service matches girl and boy threads with matching compatibility codes so they can exchange phone numbers. A thread blocks until an appropriate partner arrives. The complexity is exchanging phone numbers in the monitor because of the mutual-exclusion property. For signal scheduling, the exchange condition is necessary to block the thread finding the match, while the matcher unblocks to take the opposite number, post its phone number, and unblock the partner. For signal-block scheduling, the implicit urgent-queue replaces the explict exchange-condition and signal_block puts the finding thread on the urgent condition and unblocks the matcher.

The dating service is an example of a monitor that cannot be written using external scheduling because it requires knowledge of calling parameters to make scheduling decisions, and parameters of waiting threads are unavailable; as well, an arriving thread may not find a partner and must wait, which requires a condition variable, and condition variables imply internal scheduling.

Both internal and external scheduling extend to multiple monitors in a natural way.

```
monitor M { condition e; ... };
void foo( M & mutex m1, M & mutex m2 ) {
    ... wait( e ); ...   // wait( e, m1, m2 )
    ... wait( e, m1 ); ...
    ... wait( e, m2 ); ...
}
```

```
void rtn_1( M & mutex m1, M & mutex m2 );
void rtn_2( M & mutex m1 );
void bar( M & mutex m1, M & mutex m2 ) {
    ... waitfor( rtn ); ...      // waitfor( rtn_1, m1, m2 )
    ... waitfor( rtn, m1 ); ... // waitfor( rtn_2, m1 )
}
```

For wait(e), the default semantics is to atomically block the signaller and release all acquired mutex types in the parameter list, *ie*, wait(e, m1, m2). To override the implicit multi-monitor wait, specific mutex parameter(s) can be specified, *eg*, wait(e, m1). Wait statically verifies the released monitors are the acquired mutex-parameters so unconditional release is safe. Finally, a signaller,

```
void baz( M & mutex m1, M & mutex m2 ) {
    ... signal( e ); ...
}
```

must have acquired at least the same locks as the waiting thread signalled from the condition queue.

Similarly, for **waitfor**(rtn), the default semantics is to atomically block the acceptor and release all acquired mutex types in the parameter list, *ie*, **waitfor**(rtn, m1, m2). To override the implicit multi-monitor wait, specific mutex parameter(s) can be specified, *eg*, **waitfor**(rtn, m1). **waitfor** statically verifies the released monitors are the same as the acquired mutex-parameters of the given routine or routine pointer. To statically verify the released monitors match with the accepted routine's mutex parameters, the routine (pointer) prototype must be accessible. Overloaded routines can be disambiguated using a cast:

```
void rtn( M & mutex m );
int rtn( M & mutex m );
waitfor( (int (*)( M & mutex ))rtn, m );
```

The ability to release a subset of acquired monitors can result in a *nested monitor*[50] deadlock.

```
void foo( M & mutex m1, M & mutex m2 ) {
    ... wait( e, m1 ); ...                    // release m1, keeping m2 acquired )
void bar( M & mutex m1, M & mutex m2 ) {      // must acquire m1 and m2 )
    ... signal( e ); ...
```

The wait only releases m1 so the signalling thread cannot acquire both m1 and m2 to enter bar to get to the signal. While deadlock issues can occur with multiple/nesting acquisition, this issue results from the fact that locks, and by extension monitors, are not perfectly composable.

Finally, an important aspect of monitor implementation is barging, *ie*, can calling threads barge ahead of signalled threads? If barging is allowed, synchronization between a signaller and signallee is difficult, often requiring multiple unblock/block cycles (looping around a wait rechecking if a condition is met). In fact, signals-as-hints is completely opposite from that proposed by Hoare in the seminal paper on monitors:

> However, we decree that a signal operation be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program. It is only in this way that a waiting program has an absolute guarantee that it can acquire the resource just released by the signalling program without any danger that a third program will interpose a monitor entry and seize the resource instead.[37, p. 550]

C∀ scheduling *precludes* barging, which simplifies synchronization among threads in the monitor and increases correctness. Furthermore, C∀ concurrency has no spurious wakeup[9, § 9], which eliminates an implict form of barging. For example, there are no loops in either bounded buffer solution in Figure 6 . Supporting barging prevention as well as extending internal scheduling to multiple monitors is the main source of complexity in the design and implementation of C∀ concurrency.

6.1 | Barging Prevention

Figure 8 shows C∀ code where bulk acquire adds complexity to the internal-signalling semantics. The complexity begins at the end of the inner **mutex** statement, where the semantics of internal scheduling need to be extended for multiple monitors. The problem is that bulk acquire is used in the inner **mutex** statement where one of the monitors is already acquired. When the signalling thread reaches the end of the inner **mutex** statement, it should transfer ownership of m1 and m2 to the waiting threads to prevent barging into the outer **mutex** statement by another thread. However, both the signalling and waiting thread W1 still need monitor m1.

One scheduling solution is for the signaller to keep ownership of all locks until the last lock is ready to be transferred, because this semantics fits most closely to the behaviour of single-monitor scheduling. However, Figure 8 (C) shows this solution is complex depending on other waiters, resulting in options when the signaller finishes the inner mutex-statement. The signaller can retain m2 until completion of the outer mutex statement and pass the locks to waiter W1, or it can pass m2 to waiter W2 after completing the inner mutex-statement, while continuing to hold m1. In the latter case, waiter W2 must eventually pass m2 to waiter W1, which is complex because W1 may have waited before W2, so W2 is unaware of it. Furthermore, there is an execution sequence where the signaller always finds waiter W2, and hence, waiter W1 starves.

While a number of approaches were examined[53, § 4.3], the solution chosen for C∀ is a novel techique called *partial signalling*. Signalled threads are moved to the urgent queue and the waiter at the front defines the set of monitors necessary for it to unblock. Partial signalling transfers ownership of monitors to the front waiter. When the signaller thread exits or waits in the monitor, the

```
monitor M m1, m2;
condition c;
mutex( m1 ) { // outer
    ...
    mutex( m1, m2 ) { // inner
        ... signal( c ); ...
        // m1, m2 acquired
    } // release m2
    // m1 acquired
} // release m1
```

(A) Signalling Thread

```
mutex( m1 ) {
    ...
    mutex( m1, m2 ) {
        ... wait( c ); // block and release m1, m2
        // m1, m2 acquired
    } // release m2
    // m1 acquired
} // release m1
```

(B) Waiting Thread (W1)

```
mutex( m2 ) {
    ... wait( c ); ...
    // m2 acquired
} // release m2
```

(C) Waiting Thread (W2)

FIGURE 8 Barging Prevention

front waiter is unblocked if all its monitors are released. The benefit is encapsulating complexity into only two actions: passing monitors to the next owner when they should be released and conditionally waking threads if all conditions are met.

6.2 | Loose Object Definitions

In an object-oriented programming-language, a class includes an exhaustive list of operations. However, new members can be added via static inheritance or dynamic members, *eg*, JavaScript[54]. Similarly, monitor routines can be added at any time in C∀, making it less clear for programmers and more difficult to implement.

```
monitor M { ... };
void f( M & mutex m );
void g( M & mutex m ) { waitfor( f ); }          // clear which f
void f( M & mutex m, int );                      // different f
void h( M & mutex m ) { waitfor( f ); }          // unclear which f
```

Hence, the cfa-code for entering a monitor looks like:

```
if ( monitor is free ) // enter
else if ( already own monitor ) // continue
else if ( monitor accepts me ) // enter
else // block
```

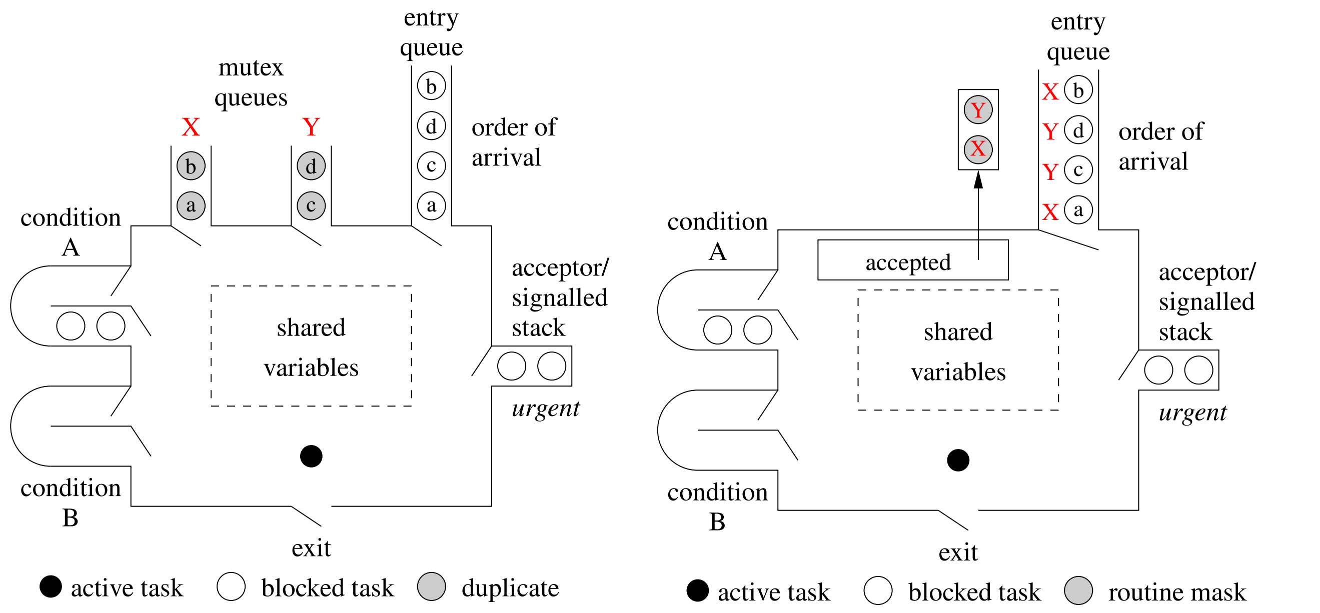
For the first two conditions, it is easy to implement a check that can evaluate the condition in a few instructions. However, a fast check for *monitor accepts me* is much harder to implement depending on the constraints put on the monitors. Figure 9 (A) shows monitors are often expressed as an entry (calling) queue, some acceptor queues, and an urgent stack/queue.

For a fixed (small) number of mutex routines (*eg*, 128), the accept check reduces to a bitmask of allowed callers, which can be checked with a single instruction. This approach requires a unique dense ordering of routines with a small upper-bound and the ordering must be consistent across translation units. For object-oriented languages these constraints are common, but C∀ mutex routines can be added in any scope and are only visible in certain translation unit, precluding program-wide dense-ordering among mutex routines.

Figure 9 (B) shows the C∀ monitor implementation. The mutex routine called is associated with each thread on the entry queue, while a list of acceptable routines is kept separately. The accepted list is a variable-sized array of accepted routine pointers, so the single instruction bitmask comparison is replaced by dereferencing a pointer followed by a (usually short) linear search.

6.3 | Multi-Monitor Scheduling

External scheduling, like internal scheduling, becomes significantly more complex for multi-monitor semantics. Even in the simplest case, new semantics needs to be established.



(A) Classical monitor **(B)** Bulk acquire monitor

FIGURE 9 Monitor Implementation

```
monitor M { ... };
void f( M & mutex m1 );
void g( M & mutex m1, M & mutex m2 ) {
    waitfor( f );                          // pass m1 or m2 to f?
}
```

The solution is for the programmer to disambiguate:

```
    waitfor( f, m2 );                      // wait for call to f with argument m2
```

Both locks are acquired by routine g, so when routine f is called, the lock for monitor m2 is passed from g to f, while g still holds lock m1. This behaviour can be extended to the multi-monitor **waitfor** statement.

```
monitor M { ... };
void f( M & mutex m1, M & mutex m2 );
void g( M & mutex m1, M & mutex m2 ) {
    waitfor( f, m1, m2 );                  // wait for call to f with arguments m1 and m2
}
```

Again, the set of monitors passed to the **waitfor** statement must be entirely contained in the set of monitors already acquired by the accepting routine. Also, the order of the monitors in a **waitfor** statement is unimportant.

Figure 10 shows an example where, for internal and external scheduling with multiple monitors, a signalling or accepting thread must match exactly, *ie*, partial matching results in waiting. For both examples, the set of monitors is disjoint so unblocking is impossible.

6.4 | Extended waitfor

Figure 11 show the extended form of the **waitfor** statement to conditionally accept one of a group of mutex routines, with a specific action to be performed *after* the mutex routine finishes. For a **waitfor** clause to be executed, its **when** must be true and an outstanding call to its corresponding member(s) must exist. The *conditional-expression* of a **when** may call a routine, but the routine must not block or context switch. If there are multiple acceptable mutex calls, selection occurs top-to-bottom (prioritized) in the **waitfor** clauses, whereas some programming languages with similar mechanisms accept non-deterministically for this case, *eg*, Go **select**. If some accept guards are true and there are no outstanding calls to these members, the acceptor is accept-blocked until a call to one of these members is made. If all the accept guards are false, the statement does nothing, unless there

```
monitor M1 {} m11, m12;                     | monitor M1 {} m11, m12;
monitor M2 {} m2;                           | monitor M2 {} m2;
condition c;                                |
void f( M1 & mutex m1, M2 & mutex m2 ) {    | void f( M1 & mutex m1, M2 & mutex m2 ) {
    signal( c );                            |
}                                           | }
void g( M1 & mutex m1, M2 & mutex m2 ) {    | void g( M1 & mutex m1, M2 & mutex m2 ) {
    wait( c );                              |     waitfor( f, m1, m2 );
}                                           | }
g( m11, m2 ); // block on wait              | g( m11, m2 ); // block on accept
f( m12, m2 ); // cannot fulfil              | f( m12, m2 ); // cannot fulfil
```

FIGURE 10 Unmatched **mutex** sets

```
when ( conditional-expression )             // optional guard
    waitfor( mutex-member-name )
        statement                           // action after call
or when ( conditional-expression )          // optional guard
    waitfor( mutex-member-name )
        statement                           // action after call
or  ...                                     // list of waitfor clauses
when ( conditional-expression )             // optional guard
    timeout                                 // optional terminating timeout clause
        statement                           // action after timeout
when ( conditional-expression )             // optional guard
    else                                    // optional terminating clause
        statement                           // action when no immediate calls
```

FIGURE 11 Extended **waitfor**

is a terminating **else** clause with a true guard, which is executed instead. Hence, the terminating **else** clause allows a conditional attempt to accept a call without blocking. If there is a **timeout** clause, it provides an upper bound on waiting. If both a **timeout** clause and an **else** clause are present, the **else** must be conditional, or the **timeout** is never triggered. In all cases, the statement following is executed *after* a clause is executed to know which of the clauses executed.

Note, a group of conditional **waitfor** clauses is *not* the same as a group of **if** statements, e.g.:

```
if ( C1 ) waitfor( mem1 );          when ( C1 ) waitfor( mem1 );
else if ( C2 ) waitfor( mem2 );     or when ( C2 ) waitfor( mem2 );
```

The left example accepts only mem1 if C1 is true or only mem2 if C2 is true. The right example accepts either mem1 or mem2 if C1 and C2 are true.

An interesting use of **waitfor** is accepting the **mutex** destructor to know when an object is deallocated.

```
void insert( Buffer(T) & mutex buffer, T elem ) with( buffer ) {
    if ( count == 10 )
        waitfor( remove, buffer ) {
            // insert elem into buffer
        } or waitfor( ^?{}, buffer ) throw insertFail;
}
```

When the buffer is deallocated, the current waiter is unblocked and informed, so it can perform an appropriate action. However, the basic **waitfor** semantics do not support this functionality, since using an object after its destructor is called is undefined. Therefore, to make this useful capability work, the semantics for accepting the destructor is the same as signal, *ie*, the call to the destructor is placed on the urgent queue and the acceptor continues execution, which throws an exception to the acceptor and then the caller is unblocked from the urgent queue to deallocate the object. Accepting the destructor is an idiomatic way to terminate a thread in C∀.

6.5 | mutex Threads

Threads in C∀ are monitors to allow direct communication among threads, *ie*, threads can have mutex routines that are called by other threads. Hence, all monitor features are available when using threads. The following shows an example of two threads directly calling each other and accepting calls from each other in a cycle.

```
thread Ping {} pi;
thread Pong {} po;
void ping( Ping & mutex ) {}
void pong( Pong & mutex ) {}
int main() {}
void main( Ping & pi ) {                    void main( Pong & po ) {
    for ( int i = 0; i < 10; i += 1 ) {         for ( int i = 0; i < 10; i += 1 ) {
        waitfor( ping, pi );                        ping( pi );
        pong( po );                                 waitfor( pong, po );
    }                                           }
}                                           }
```

Note, the ping/pong threads are globally declared, pi/po, and hence, start (and possibly complete) before the program main starts.

6.6 | Low-level Locks

For completeness and efficiency, C∀ provides a standard set of low-level locks: recursive mutex, condition, semaphore, barrier, *etc*., and atomic instructions: fetchAssign, fetchAdd, testSet, compareSet, *etc*.

7 | PARALLELISM

Historically, computer performance was about processor speeds. However, with heat dissipation being a direct consequence of speed increase, parallelism is the new source for increased performance[18,19]. Now, high-performance applications must care about parallelism, which requires concurrency. The lowest-level approach of parallelism is to use *kernel threads* in combination with semantics like fork, join, *etc*. However, kernel threads are better as an implementation tool because of complexity and higher cost. Therefore, different abstractions are often layered onto kernel threads to simplify them, *eg*, pthreads.

7.1 | User Threads with Preemption

A direct improvement on kernel threads is user threads, *eg*, Erlang[7] and μC++[55]. This approach provides an interface that matches the language paradigms, more control over concurrency by the language runtime, and an abstract (and portable) interface to the underlying kernel threads across operating systems. In many cases, user threads can be used on a much larger scale (100,000 threads). Like kernel threads, user threads support preemption, which maximizes nondeterminism, but introduces the same concurrency errors: race, livelock, starvation, and deadlock. C∀ adopts user-threads as they represent the truest realization of concurrency and can build any other concurrency approach, *eg*, thread pools and actors[56].

7.2 | User Threads without Preemption (Fiber)

A variant of user thread is *fibers*, which removes preemption, *eg*, Go[6] goroutines. Like functional programming, which removes mutation and its associated problems, removing preemption from concurrency reduces nondeterminism, making race and deadlock errors more difficult to generate. However, preemption is necessary for concurrency that relies on spinning, so there are a class of problems that cannot be programmed without preemption.

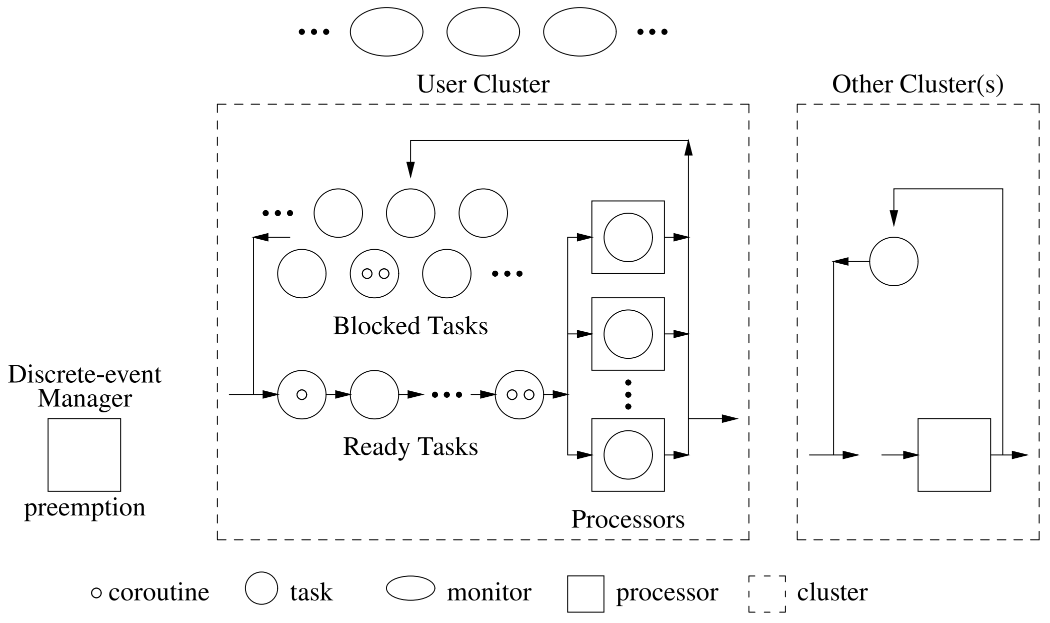


FIGURE 12 C∀ Runtime Structure

7.3 | Thread Pools

In contrast to direct threading is indirect *thread pools*, where small jobs (work units) are inserted into a work pool for execution. If the jobs are dependent, *ie*, interact, there is an implicit/explicit dependency graph that ties them together. While removing direct concurrency, and hence the amount of context switching, thread pools significantly limit the interaction that can occur among jobs. Indeed, jobs should not block because that also blocks the underlying thread, which effectively means the CPU utilization, and therefore throughput, suffers. While it is possible to tune the thread pool with sufficient threads, it becomes difficult to obtain high throughput and good core utilization as job interaction increases. As well, concurrency errors return, which threads pools are suppose to mitigate.

8 | C∀ RUNTIME STRUCTURE

Figure 12 illustrates the runtime structure of a C∀ program. In addition to the new kinds of objects introduced by C∀, there are two more runtime entities used to control parallel execution: cluster and (virtual) processor. An executing thread is illustrated by its containment in a processor.

8.1 | Cluster

A *cluster* is a collection of threads and virtual processors (abstract kernel-thread) that execute the threads (like a virtual machine). The purpose of a cluster is to control the amount of parallelism that is possible among threads, plus scheduling and other execution defaults. The default cluster-scheduler is single-queue multi-server, which provides automatic load-balancing of threads on processors. However, the scheduler is pluggable, supporting alternative schedulers. If several clusters exist, both threads and virtual processors, can be explicitly migrated from one cluster to another. No automatic load balancing among clusters is performed by C∀.

When a C∀ program begins execution, it creates a user cluster with a single processor and a special processor to handle preemption that does not execute user threads. The user cluster is created to contain the application user-threads. Having all threads execute on the one cluster often maximizes utilization of processors, which minimizes runtime. However, because of limitations of the underlying operating system, heterogeneous hardware, or scheduling requirements (real-time), multiple clusters are sometimes necessary.

8.2 | Virtual Processor

A virtual processor is implemented by a kernel thread (*eg*, UNIX process), which is subsequently scheduled for execution on a hardware processor by the underlying operating system. Programs may use more virtual processors than hardware processors. On a multiprocessor, kernel threads are distributed across the hardware processors resulting in virtual processors executing in parallel. (It is possible to use affinity to lock a virtual processor onto a particular hardware processor[57,58,59,60,61], which is used when caching issues occur or for heterogeneous hardware processors.) The C∀ runtime attempts to block unused processors and unblock processors as the system load increases; balancing the workload with processors is difficult. Preemption occurs on virtual processors rather than user threads, via operating-system interrupts. Thus virtual processors execute user threads, where preemption frequency applies to a virtual processor, so preemption occurs randomly across the executed user threads. Turning off preemption transforms user threads into fibers.

8.3 | Debug Kernel

There are two versions of the C∀ runtime kernel: debug and non-debug. The debugging version has many runtime checks and internal assertions, *eg*, stack (non-writable) guard page, and checks for stack overflow whenever context switches occur among coroutines and threads, which catches most stack overflows. After a program is debugged, the non-debugging version can be used to decrease space and increase performance.

9 | IMPLEMENTATION

Currently, C∀ has fixed-sized stacks, where the stack size can be set at coroutine/thread creation but with no subsequent growth. Schemes exist for dynamic stack-growth, such as stack copying and chained stacks. However, stack copying requires pointer adjustment to items on the stack, which is impossible without some form of garbage collection. As well, chained stacks require all modules be recompiled to use this feature, which breaks backward compatibility with existing C libraries. In the long term, it is likely C libraries will migrate to stack chaining to support concurrency, at only a minimal cost to sequential programs. Nevertheless, experience teaching μC++[62] shows fixed-sized stacks are rarely an issue in most concurrent programs.

A primary implementation challenge is avoiding contention from dynamically allocating memory because of bulk acquire, *eg*, the internal-scheduling design is (almost) free of allocations. All blocking operations are made by parking threads onto queues, therefore all queues are designed with intrusive nodes, where each node has preallocated link fields for chaining. Furthermore, several bulk-acquire operations need a variable amount of memory. This storage is allocated at the base of a thread's stack before blocking, which means programmers must add a small amount of extra space for stacks.

In C∀, ordering of monitor acquisition relies on memory ordering to prevent deadlock[63], because all objects have distinct non-overlapping memory layouts, and mutual-exclusion for a monitor is only defined for its lifetime. When a mutex call is made, pointers to the concerned monitors are aggregated into a variable-length array and sorted. This array persists for the entire duration of the mutual exclusion and is used extensively for synchronization operations.

To improve performance and simplicity, context switching occurs inside a routine call, so only callee-saved registers are copied onto the stack and then the stack register is switched; the corresponding registers are then restored for the other context. Note, the instruction pointer is untouched since the context switch is always inside the same routine. Unlike coroutines, threads do not context switch among each other; they context switch to the cluster scheduler. This method is a 2-step context-switch and provides a clear distinction between user and kernel code, where scheduling and other system operations happen. The alternative 1-step context-switch uses the *from* thread's stack to schedule and then context-switches directly to the *to* thread's stack. Experimental results (not presented) show the performance of these two approaches is virtually equivalent, because both approaches are dominated by locking to prevent a race condition.

All kernel threads (pthreads) created a stack. Each C∀ virtual processor is implemented as a coroutine and these coroutines run directly on the kernel-thread stack, effectively stealing this stack. The exception to this rule is the program main, *ie*, the initial kernel thread that is given to any program. In order to respect C expectations, the stack of the initial kernel thread is used by program main rather than the main processor, allowing it to grow dynamically as in a normal C program.

Finally, an important aspect for a complete threading system is preemption, which introduces extra non-determinism via transparent interleaving, rather than cooperation among threads for proper scheduling and processor fairness from long-running threads. Because preemption frequency is usually long (1 millisecond) performance cost is negligible. Preemption is normally

TABLE 1 Experiment environment

Architecture	x86_64	NUMA node(s)	8
CPU op-mode(s)	32-bit, 64-bit	Model name	AMD Opteron™ Processor 6380
Byte Order	Little Endian	CPU Freq	2.5 GHz
CPU(s)	64	L1d cache	16 KiB
Thread(s) per core	2	L1i cache	64 KiB
Core(s) per socket	8	L2 cache	2048 KiB
Socket(s)	4	L3 cache	6144 KiB
Operating system	Ubuntu 16.04.3 LTS	Kernel	Linux 4.4-97-generic
gcc	6.3	C∀	1.0.0
Java	OpenJDK-9	Go	1.9.2

handled by setting a count-down timer on each virtual processor. When the timer expires, an interrupt is delivered, and the interrupt handler resets the count-down timer, and if the virtual processor is executing in user code, the signal handler performs a user-level context-switch, or if executing in the language runtime-kernel, the preemption is ignored or rolled forward to the point where the runtime kernel context switches back to user code. Multiple signal handlers may be pending. When control eventually switches back to the signal handler, it returns normally, and execution continues in the interrupted user thread, even though the return from the signal handler may be on a different kernel thread than the one where the signal is delivered. The only issue with this approach is that signal masks from one kernel thread may be restored on another as part of returning from the signal handler; therefore, the same signal mask is required for all virtual processors in a cluster.

However, on current UNIX systems:

> A process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked. If more than one of the threads has the signal unblocked, then the kernel chooses an arbitrary thread to which to deliver the signal. SIGNAL(7) - Linux Programmer's Manual

Hence, the timer-expiry signal, which is generated *externally* by the UNIX kernel to the UNIX process, is delivered to any of its UNIX subprocesses (kernel threads). To ensure each virtual processor receives its own preemption signals, a discrete-event simulation is run on a special virtual processor, and only it sets and receives timer events. Virtual processors register an expiration time with the discrete-event simulator, which is inserted in sorted order. The simulation sets the count-down timer to the value at the head of the event list, and when the timer expires, all events less than or equal to the current time are processed. Processing a preemption event sends an *internal* SIGUSR1 signal to the registered virtual processor, which is always delivered to that processor.

10 | PERFORMANCE

To verify the implementation of the C∀ runtime, a series of microbenchmarks are performed comparing C∀ with other widely used programming languages with concurrency. Table 1 shows the specifications of the computer used to run the benchmarks, and the versions of the software used in the comparison.

All benchmarks are run using the following harness:

```
unsigned int N = 10_000_000;
#define BENCH( run ) Time before = getTimeNsec(); run; Duration result = (getTimeNsec() - before) / N;
```

The method used to get time is clock_gettime(CLOCK_REALTIME). Each benchmark is performed N times, where N varies depending on the benchmark; the total time is divided by N to obtain the average time for a benchmark. All omitted tests for other languages are functionally identical to the shown C∀ test.

Context-Switching

In procedural programming, the cost of a routine call is important as modularization (refactoring) increases. (In many cases, a compiler inlines routine calls to eliminate this cost.) Similarly, when modularization extends to coroutines/tasks, the time for a context switch becomes a relevant factor. The coroutine context-switch is 2-step using resume/suspend, *ie*, from resumer to suspender and from suspender to resumer. The thread context switch is 2-step using yield, *ie*, enter and return from the runtime kernel. Figure 13 shows the code for coroutines/threads with all results in Table 2 . The difference in performance between coroutine and thread context-switch is the cost of scheduling for threads, whereas coroutines are self-scheduling.

Mutual-Exclusion

Mutual exclusion is measured by entering/leaving a critical section. For monitors, entering and leaving a monitor routine is measured. Figure 14 shows the code for C∀ with all results in Table 3 . To put the results in context, the cost of entering a non-inline routine and the cost of acquiring and releasing a `pthread_mutex` lock is also measured. Note, the incremental cost of bulk acquire for C∀, which is largely a fixed cost for small numbers of mutex objects.

Internal Scheduling

Internal scheduling is measured by waiting on and signalling a condition variable. Figure 15 shows the code for C∀, with results in Table 4 . Note, the incremental cost of bulk acquire for C∀, which is largely a fixed cost for small numbers of mutex objects. Java scheduling is significantly greater because the benchmark explicitly creates multiple thread in order to prevent the JIT from making the program sequential, *ie*, removing all locking.

External Scheduling

External scheduling is measured by accepting a call using the **waitfor** statement (_Accept in μC++). Figure 16 shows the code for C∀, with results in Table 5 . Note, the incremental cost of bulk acquire for C∀, which is largely a fixed cost for small numbers of mutex objects.

Object Creation

Object creation is measured by creating/deleting the specific kind of concurrent object. Figure 17 shows the code for C∀, with results in Table 6 . The only note here is that the call stacks of C∀ coroutines are lazily created, therefore without priming the coroutine to force stack creation, the creation cost is artificially low.

11 | CONCLUSION

This paper demonstrates a concurrency API that is simple, efficient, and able to build higher-level concurrency features. The approach provides concurrency based on a preemptive M:N user-level threading-system, executing in clusters, which encapsulate scheduling of work on multiple kernel threads providing parallelism. The M:N model is judged to be efficient and provide greater flexibility than a 1:1 threading model. High-level objects (monitor/task) are the core mechanism for mutual exclusion and synchronization. A novel aspect is allowing multiple mutex-objects to be accessed simultaneously reducing the potential for deadlock for this complex scenario. These concepts and the entire C∀ runtime-system are written in the C∀ language, demonstrating the expressiveness of the C∀ language. Performance comparisons with other concurrent systems/languages show the C∀ approach is competitive across all low-level operations, which translates directly into good performance in well-written concurrent applications. C programmers should feel comfortable using these mechanisms for developing concurrent applications, with the ability to obtain maximum available performance by mechanisms at the appropriate level.

12 | FUTURE WORK

While concurrency in C∀ has a strong start, development is still underway and there are missing features.

Flexible Scheduling

An important part of concurrency is scheduling. Different scheduling algorithms can affect performance (both in terms of average and variation). However, no single scheduler is optimal for all workloads and therefore there is value in being able to change

```
coroutine C {} c;
void main( C & ) { for ( ;; ) { suspend(); } }
int main() {
    BENCH(
        for ( size_t i = 0; i < N; i += 1 ) { resume( c ); } )
    sout | result`ns;
}
```

(A) Coroutine

```
int main() {
    BENCH(
        for ( size_t i = 0; i < N; i += 1 ) { yield(); } )
    sout | result`ns;
}
```

(B) Thread

FIGURE 13 C∀ context-switch benchmark

TABLE 2 Context switch comparison (nanoseconds)

	Median	Average	Std Dev
Kernel Thread	333.5	332.96	4.1
C∀ Coroutine	49	48.68	0.47
C∀ Thread	105	105.57	1.37
μC++ Coroutine	44	44	0
μC++ Thread	100	99.29	0.96
Goroutine	145	147.25	4.15
Java Thread	373.5	375.14	8.72

```
monitor M { ... } m1/*, m2, m3, m4*/;
void __attribute__((noinline)) do_call( M & mutex m/*, m2, m3, m4*/ ) {}
int main() {
    BENCH( for( size_t i = 0; i < N; i += 1 ) { do_call( m1/*, m2, m3, m4*/ ); } )
    sout | result`ns;
}
```

FIGURE 14 C∀ acquire/release mutex benchmark

TABLE 3 Mutex comparison (nanoseconds)

	Median	Average	Std Dev
C routine	2	2	0
FetchAdd + FetchSub	26	26	0
Pthreads Mutex Lock	31	31.71	0.97
μC++ **monitor** member routine	31	31	0
C∀ **mutex** routine, 1 argument	46	46.68	0.93
C∀ **mutex** routine, 2 argument	84	85.36	1.99
C∀ **mutex** routine, 4 argument	158	161	4.22
Java synchronized routine	27.5	29.79	2.93

```
volatile int go = 0;
condition c;
monitor M { ... } m;
void __attribute__((noinline)) do_call( M & mutex a1 ) { signal( c ); }
thread T {};
void main( T & this ) {
    while ( go == 0 ) { yield(); }     // wait for other thread to start
    while ( go == 1 ) { do_call( m ); }
}
int  __attribute__((noinline)) do_wait( M & mutex m ) {
    go = 1;   // continue other thread
    BENCH( for ( size_t i = 0; i < N; i += 1 ) { wait( c ); } );
    go = 0;   // stop other thread
    sout | result`ns;
}
int main() {
    T t;
    do_wait( m );
}
```

FIGURE 15 C∀ Internal-scheduling benchmark

TABLE 4 Internal-scheduling comparison (nanoseconds)

	Median	Average	Std Dev
Pthreads Condition Variable	6005	5681.43	835.45
μC++ signal	324	325.54	3, 02
C∀ signal, 1 **monitor**	368.5	370.61	4.77
C∀ signal, 2 **monitor**	467	470.5	6.79
C∀ signal, 4 **monitor**	700.5	702.46	7.23
Java notify	15471	172511	5689

the scheduler for given programs. One solution is to offer various tweaking options, allowing the scheduler to be adjusted to the requirements of the workload. However, to be truly flexible, a pluggable scheduler is necessary. Currently, the C∀ pluggable scheduler is too simple to handle complex scheduling, *eg*, quality of service and real-time, where the scheduler must interact with mutex objects to deal with issues like priority inversion.

Non-Blocking I/O

Many modern workloads are not bound by computation but IO operations, a common case being web servers and XaaS[64] (anything as a service). These types of workloads require significant engineering to amortizing costs of blocking IO-operations. At its core, non-blocking I/O is an operating-system level feature queuing IO operations, *eg*, network operations, and registering for notifications instead of waiting for requests to complete. Current trends use asynchronous programming like callbacks, futures, and/or promises, *eg*, Node.js[65] for JavaScript, Spring MVC[66] for Java, and Django[67] for Python. However, these solutions lead to code that is hard to create, read and maintain. A better approach is to tie non-blocking I/O into the concurrency system to provide ease of use with low overhead, *eg*, thread-per-connection web-services. A non-blocking I/O library is currently under development for C∀.

Other Concurrency Tools

While monitors offer flexible and powerful concurrency for C∀, other concurrency tools are also necessary for a complete multi-paradigm concurrency package. Examples of such tools can include futures and promises[68], executors and actors. These additional features are useful when monitors offer a level of abstraction that is inadequate for certain tasks. As well, new C∀ extensions should make it possible to create a uniform interface for virtually all mutual exclusion, including monitors and low-level locks.

```
volatile int go = 0;
monitor M { ... } m;
thread T {};
void __attribute__((noinline)) do_call( M & mutex ) {}
void main( T & ) {
    while ( go == 0 ) { yield(); }    // wait for other thread to start
    while ( go == 1 ) { do_call( m ); }
}
int __attribute__((noinline)) do_wait( M & mutex m ) {
    go = 1;   // continue other thread
    BENCH( for ( size_t i = 0; i < N; i += 1 ) { waitfor( do_call, m ); } )
    go = 0;   // stop other thread
    sout | result`ns;
}
int main() {
    T t;
    do_wait( m );
}
```

FIGURE 16 C∀ external-scheduling benchmark

TABLE 5 External-scheduling comparison (nanoseconds)

	Median	Average	Std Dev
μC++ _Accept	358	359.11	2.53
C∀ **waitfor**, 1 **monitor**	359	360.93	4.07
C∀ **waitfor**, 2 **monitor**	450	449.39	6.62
C∀ **waitfor**, 4 **monitor**	652	655.64	7.73

```
thread MyThread {};
void main( MyThread & ) {}
int main() {
    BENCH( for ( size_t i = 0; i < N; i += 1 ) { MyThread m; } )
    sout | result`ns;
}
```

FIGURE 17 C∀ object-creation benchmark

TABLE 6 Creation comparison (nanoseconds)

	Median	Average	Std Dev
Pthreads	28091	28073.39	163.1
C∀ Coroutine Lazy	6	6.07	0.26
C∀ Coroutine Eager	520	520.61	2.04
C∀ Thread	2032	2016.29	112.07
μC++ Coroutine	106	107.36	1.47
μC++ Thread	536.5	537.07	4.64
Goroutine	3103	3086.29	90.25
Java Thread	103416.5	103732.29	1137

Implicit Threading

Basic concurrent (embarrassingly parallel) applications can benefit greatly from implicit concurrency, where sequential programs are converted to concurrent, possibly with some help from pragmas to guide the conversion. This type of concurrency can be achieved both at the language level and at the library level. The canonical example of implicit concurrency is concurrent nested **for** loops, which are amenable to divide and conquer algorithms[55]. The C∀ language features should make it possible to develop a reasonable number of implicit concurrency mechanism to solve basic HPC data-concurrency problems. However, implicit concurrency is a restrictive solution with significant limitations, so it can never replace explicit concurrent programming.

13 | ACKNOWLEDGEMENTS

The authors would like to recognize the design assistance of Aaron Moss, Rob Schluntz and Andrew Beach on the features described in this paper. Funding for this project has been provided by Huawei Ltd. (http://www.huawei.com), and Peter Buhr is partially funded by the Natural Sciences and Engineering Research Council of Canada.

References

1. Hochstein Lorin, Carver Jeff, Shull Forrest, et al. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In: *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference* :35-35. IEEE; 2005.
2. Thread Building Blocks Intel, https://www.threadingbuildingblocks.org.
3. OpenMP Application Program Interface, Version 4.5 . 2015. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.
4. Gosling James, Joy Bill, Steele Guy, Bracha Gilad. *The Java Language Specification*. Reading: Addison-Wesley; 2nd ed. 2000.
5. Hoare C. A. R.. Communicating Sequential Processes. *Communications ACM*. 1978; 21(8):666-677.
6. Griesemer Robert, Pike Rob, Thompson Ken. Go Programming Language. Google . 2009. http://golang.org/ref/spec.
7. Erlang AB. Erlang/OTP System Documentation 8.1 . 2016. http://erlang.org/doc/pdf/otp-system-documentation.pdf.
8. Message Passing Interface Forum. University of Tennessee, Knoxville, TennesseeMPI: A Message-Passing Interface Standard, Version 3.1 . 2015. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.
9. Buhr Peter A., Harji Ashif S.. Concurrent Urban Legends. *Concurrency Comput.: Pract. Exper.*. 2005; 17(9):1133-1172.
10. C∀ Features . . https://plg.uwaterloo.ca/~cforall/features.
11. Moss Aaron, Schluntz Robert, Buhr Peter A.. C∀ : Adding Modern Programming Language Features to C. *Softw. Pract. Exper.*. 2018; 48(12):2111-2146. http://dx.doi.org/10.1002/spe.2624.
12. C Programming Language ISO/IEC 9889:2011-12. https://www.iso.org/standard/57853.html. 3rd ed. . 2012.
13. Jensen Kathleen, Wirth Niklaus. *Pascal User Manual and Report, ISO Pascal Standard*. Springer–Verlag; 4th ed. 1991. Revised by Andrew B. Mickel and James F. Miner.
14. Ada . The Programming Language Ada: Reference Manual. United States Department of Defense. ANSI/MIL-STD-1815A-1983 ed. . 1983. Springer, New York.
15. Conway Melvin E.. Design of a Separable Transition-Diagram Compiler. *Communications ACM*. 1963; 6(7):396-408.
16. Marlin Christopher D.. *Coroutines: A Programming Methodology, a Language Design and an Implementation* Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis, vol. 95: . New York: Springer; 1980.
17. Python . Python Language Reference, Release 3.7.2. Python Software Foundation. https://docs.python.org/3/reference/index.html . 2018.
18. Sutter Herb. A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal : Software Tools for the Professional Programmer*. 2005; 30(3):16-22.
19. Sutter Herb, Larus James. Software and the Concurrency Revolution. *Queue*. 2005; 3(7):54-62.
20. Butenhof David R.. *Programming with POSIX Threads*. Professional ComputingBoston: Addison-Wesley; 1997.
21. C++ Programming Language ISO/IEC 14882:2014. https://www.iso.org/standard/64029.html. 4th ed. . 2014.
22. Microsoft Corporation. Microsoft Visual C++ .NET Language Reference . 2002. Microsoft Press, Redmond, Washington, U.S.A.

23. Kowalke Oliver. Boost Coroutine Library http://www.boost.org/doc/libs/1_61_0/libs/coroutine/doc/html/index.html . 2015.

24. ECMA International Standardizing Information and Communication Systems. C# Language Specification, Standard ECMA-334. 4th ed. . 2006.

25. Scala Language Specification, Version 2.11. École Polytechnique Fédérale de Lausanne . 2016. http://www.scala-lang.org/files/archive/-spec/2.11.

26. Hudak Paul, Fasel Joseph H.. A Gentle Introduction to Haskell. *SIGPLAN Not.*. 1992; 27(5):T1-53.

27. Lightbend Inc. Akka Scala Documentation, Release 2.4.11 . 2016. http://doc.akka.io/docs/akka/2.4/AkkaScala.pdf.

28. Cheriton D. R.. *The Thoth System: Multi-Process Structuring and Portability*. American Elsevier; 1982.

29. Gentleman W. Morven. *Using the Harmony Operating System.* 24685: National Research Council of Canada, Ottawa, Canada; 1985.

30. Cheriton David R.. The V Distributed System. *Communications ACM.* 1988; 31(3):314-333.

31. Dijkstra E. W.. The Structure of the "THE"–Multiprogramming System. *Communications ACM.* 1968; 11(5):341-346.

32. Campbell R. H., Habermann A. N.. *The Specification of Process Synchronization by Path Expressions* Lecture Notes in Computer Science, vol. 16: . Springer; 1974.

33. Herlihy Maurice, Moss J. Eliot B.. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News.* 1993; 21(2):289–300.

34. Nakaike Takuya, Odaira Rei, Gaudet Matthew, Michael Maged M., Tomari Hisanobu. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. *SIGARCH Comput. Archit. News.* 2015; 43(3):144–157.

35. International Standard ISO/IEC TS 19841:2015. http://www.iso.orgTechnical Specification for C++ Extensions for Transactional Memory . 2015.

36. Brinch Hansen Per. *Operating System Principles*. Englewood Cliffs: Prentice-Hall; 1973.

37. Hoare C. A. R.. Monitors: An Operating System Structuring Concept. *Communications ACM.* 1974; 17(10):549-557.

38. Brinch Hansen Per. The Programming Language Concurrent Pascal. *IEEE Trans. Softw. Eng.*. 1975; 2:199-206.

39. Mitchell James G., Maybury William, Sweet Richard. *Mesa Language Manual.* CSL–79–3: Xerox Palo Alto Research Center; 1979.

40. Wirth Niklaus. *Programming in Modula-2*. Texts and Monographs in Computer ScienceNew York: Springer; 4th ed. 1988.

41. Holt R. C., Cordy J. R.. The Turing Programming Language. *Communications ACM.* 1988; 31(12):1410-1423.

42. Birrell Andrew, Brown Mark R., Cardelli Luca, et al. *Systems Programming with Modula-3*. Prentice-Hall Series in Innovative TechnologyEnglewood Cliffs: Prentice-Hall; 1991.

43. Gosling James, Rosenthal David S. H., Arden Richelle J.. *The NeWS Book*. Springer-Verlag; 1989.

44. Raj Rajendra K., Tempero Ewan, Levy Henry M., Black Andrew P., Hutchinson Norman C., Jul Eric. Emerald: A General-Purpose Programming Language. *Softw. Pract. Exper.*. 1991; 21(1):91-118.

45. Buhr P. A., Ditchfield Glen, Stroobosscher R. A., Younger B. M., Zarnke C. R.. μC++: Concurrency in the Object-Oriented Language C++. *Softw. Pract. Exper.*. 1992; 22(2):137-172.

46. Dijkstra Edsger W.. *Cooperating Sequential Processes.* : Technological UniversityEindhoven, Netherlands; 1965. Reprinted in? pp. 43–112.

47. Joung Yuh-Jzer. Asynchronous group mutual exclusion. *Dist. Comput.*. 2000; 13(4):189–206.

48. Courtois P. J., Heymans F., Parnas D. L.. Concurrent Control with Readers and Writers. *Communications ACM.* 1971; 14(10):667-668.

49. Andrews Gregory R.. A Method for Solving Synronization Problems. *Science of Computer Programming.* 1989; 13(4):1-21.

50. Lister Andrew. The Problem of Nested Monitor Calls. *Operating Systems Review.* 1977; 11(3):5-7.

51. Dice Dave, Lev Yossi, Marathe Virendra J., Moir Mark, Nussbaum Dan, Olszewski Marek. Simplifying Concurrent Algorithms by Exploiting Hardware Transactional Memory. In: *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures* SPAA'10. :325-334. ACM; 2010; New York, NY, USA.

52. Buhr Peter A.. μC++ Annotated Reference Manual, Version 7.0.0. University of Waterloo . 2018. https://plg.uwaterloo.ca/~usystem/-pub/uSystem/uC++.pdf.

53. Delisle Thierry. Concurrency in C∀. Master's thesis. School of Computer Science, University of Waterloo . 2018. https://-uwspace.uwaterloo.ca/handle/10012/12888.

54. ECAM International. Rue du Rhone 114, CH-1204 Geneva, SwitzerlandECMAScript 2015 Language Specification JavaScript . 2015. 6th Edition.

55. Buhr Peter A.. *Understanding Control Flow: Concurrent Programming using μC++*. Switzerland: Springer; 2016.

56. Agha Gul A.. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge; 1986.

57. Linux man page - sched_setaffinity(2).

58. Windows (vs.85) - SetThreadAffinityMask function.

59. FreeBSD General Commands Manual - CPUSET(1).

60. NetBSD Library Functions Manual - AFFINITY(3).

61. Affinity API Release Notes for OS X v10.5.

62. CS343 https://www.student.cs.uwaterloo.ca/∼cs343 . 2018.

63. Havender J. W.. Avoiding Deadlock in Multitasking Systems. *IBM Systems J.*. 1968; 7(2):74-84.

64. Duan Yucong, Fu Guohua, Zhou Nianjun, Sun Xiaobing, Narendra Nanjangud C., Hu Bo. Everything As a Service (XaaS) on the Cloud: Origins, Current and Future Trends. In: *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing* CLOUD'15. :621–628. IEEE Computer Society; 2015; Washington, DC, USA.

65. Node.js https://nodejs.org/en/.

66. Spring Web MVC https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html.

67. Django https://www.djangoproject.com/.

68. Liskov Barbara, Shrira Liuba. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *SIGPLAN Not.*. 1988; 23(7):260-267. Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation.

69. Genuys F., ed.*Programming Languages*. London, New York: Academic Press; 1968. NATO Advanced Study Institute, Villard-de-Lans, 1966.