SHIM: A Deterministic Approach to Programming with Threads*

Olivier Tardieu and Stephen A. Edwards†
Department of Computer Science
Columbia University, New York
{tardieu,sedwards}@cs.columbia.edu

Abstract

Concurrent programming languages should be a good fit for embedded systems because they match the intrinsic parallelism of their architectures and environments. Unfortunately, most concurrent programming formalisms are prone to races and nondeterminism, despite the presence of mechanisms such as monitors.

In this paper, we propose SHIM, the core of a concurrent language with disciplined shared variables that remains deterministic, meaning the behavior of a program is independent of the scheduling of concurrent operations. SHIM does not sacrifice power or flexibility to achieve this determinism. It supports both synchronous and asynchronous paradigms—loosely and tightly synchronized threads—the dynamic creation of threads and shared variables, recursive procedures, and exceptions.

We illustrate our programming model with examples including breadth-first-search algorithms and pipelines. By construction, they are race-free. We provide the formal semantics of SHIM and a preliminary implementation.

1 Introduction

Embedded systems differ from traditional computing systems in their need for concurrent descriptions to handle simultaneous activities in their environment or to exploit heterogeneous hardware. While it would be nice to program such systems in purely sequential languages, this greatly hinders exploiting parallelism. Instead, we believe the solution is to provide fundamentally concurrent languages that by construction avoid many of the usual pitfalls of parallel programming.

We say the behavior of a system is deterministic if and only if it depends exclusively on external decisions, i.e., on well-defined inputs of the system (which may include a clock), rather than internal decisions of the compiler, optimizer, runtime scheduler, debugger, etc. The motivation for our work rests on two central assumptions. Most programs (including concurrent programs) are meant to behave deterministically. However, most programming languages (including Java and C) do not guarantee determinism but instead provide constructs designed to help achieve it.

C's nondeterminism lurks in subtle places, such as function argument evaluation order, and in "undefined behavior," such as reading uninitialized memory. Most programmers are careful enough to avoid calling functions with side-effects when passing parameters, but the undefined behavior of C produces a whole host of problems including buffer overflows, which is probably the leading cause of computer insecurity today. Languages such as Cyclone [26] and the CCured rewriter [29] have been developed to attack exactly these sources of nondeterminism in the language.

The design of Java successfully avoids most of the obviously nondeterministic aspects of C by adding array bounds checking, fixing expression evaluation order, etc., but its concurrency introduces a whole host of potential sources of nondeterminism, leading to concurrency-related bugs such as data races and the like. It is these concurrency-induced sources of nondeterminism that we are primarily concerned with avoiding.

In this work, we advocate for SHIM, a deterministic concurrent programming language. A program written in SHIM is guaranteed to behave the same regardless of the scheduling of concurrent operations. While the restrictions SHIM imposes no doubt make it impossible to implement certain algorithms that appear nondeterministic but actually are well-behaved, we believe SHIM is both expressive and amenable to efficient implementation. We argue for SHIM's expressivity through a series of examples, and demonstrate a preliminary compiler for the language able to generate single-threaded C code.

1.1 The New SHIM

Our original SHIM (Software/Hardware Integration Medium) model of computation [15, 16] provides deterministic concurrency in a simple setting: SHIM systems consist of sequential processes that communicate using rendezvous through point-to-point communication channels. SHIM systems are therefore delay-insensitive and deterministic in the same way and for the same reasons as Kahn's networks [27], but are simpler to schedule and require only bounded resources by adopting rendezvous-style communication inspired by Hoare's CSP [23].

*This is an extended version of the paper with the same name that appeared in Emsoft 2006.

†Edwards and his group are supported by an NSF CAREER award, a grant from Intel and Altera, an award from the SRC, and by New York State's NYSTAR program.

We designed the original SHIM model to capture the mix of finely scheduled hardware and coarsely scheduled software typical of embedded systems. However, the programming model we have shown [15, 16] is limited, much better at describing static hardware components than complex, dynamic software tasks.

In this paper, we present a major extension of the SHIM model and language. The result is a programming language that resembles C and Java (and can be used as such) while still guaranteeing deterministic concurrency without requiring careful attention to the use of semaphores or monitors found in many concurrent programming languages. Instead of a static network of processes connected by predefined point-to-point communication channels, SHIM now enables the dynamic creation of threads and shared variables. Using recursion, one can then instantiate arbitrarily many threads. We also introduce concurrent, deterministic exceptions. While these are meant to resemble the exceptions in Java (and have the same semantics in single-threaded code), they interact smoothly with concurrency, providing a structured, deterministic way to terminate groups of processes and thus provide a powerful tool for describing sequential behavior in a concurrent setting. We believe that such control constructs for groups of concurrent processes has been a key omission in most dataflow-oriented concurrent languages.

One attribute of SHIM is the ease with which concurrency can be introduced. For example, here is a typical sequential SHIM program that looks for a *key* in a binary *tree* depth first throwing exception *Found* if it finds the *key*.

```
depth_first_search(int key, Tree tree) {
  if (tree != null) {
    if(key == tree.key) throw Found;
    depth_first_search(key, tree.left);
    depth_first_search(key, tree.right);
  }
}
```

Adding two lines of code turns this depth-first-search algorithm into a concurrent breadth-first-search:

```
breadth_first_search(int key, Tree tree) {
  if (tree != null) {
    if(key == tree.key) throw Found;
    next key;          // synchronize concurrent threads
      breadth_first_search(key, tree.left);
    par                // fork concurrent threads
      breadth_first_search(key, tree.right);
  }
}
```

In Section 4, we augment this code to return a value associated with the key. An obvious pitfall in the concurrent version of the algorithm would be ignoring what to do when the key appears multiple times in the tree; *SHIM makes it impossible not to include an arbitration policy for this case*.

Overall, we believe its concurrency, determinism, facilities for dynamic thread creation, and exceptions makes it possible to use SHIM to program true software components for an embedded system and obtain strong functionality guarantees about complete (hardware/software) designs.

$e ::= L \mid V \mid op_1\ e \mid e\ op_2\ e \mid \texttt{(}\ e\ \texttt{)}$	**expressions**
$s ::= V\ \texttt{=}\ e\texttt{;} \mid F\texttt{(}\ (V(,V)^*)?\ \texttt{);}$	**statements**
$\quad \mid \texttt{\{}\ b^*\ \texttt{\}} \mid \texttt{if (}\ e\ \texttt{)}\ s\ \texttt{else}\ s \mid \texttt{while (}\ e\ \texttt{)}\ s$	
$\quad \mid s\ \texttt{par}\ s \mid \texttt{next}\ V\texttt{;} \mid \texttt{try}\ s\ \texttt{catch(}\ E\ \texttt{)}\ s \mid \texttt{throw}\ E\texttt{;}$	
$b ::= T\ V\texttt{;} \mid s$	**block statements**
$d ::= T\ V \mid T\ \&V$	**parameter declarations**
$p ::= P\texttt{(}\ (d(,d)^*)?\ \texttt{)}\ \texttt{\{}\ b^*\ \texttt{\}}$	**procedure declarations**
$r ::= p^*\ \texttt{main()}\ \texttt{\{}\ b^*\ \texttt{\}}$	**programs**

L denotes literals, T types (e.g., `int`, `void`), E exception identifiers, V variable identifiers, and P procedure identifiers. `par` binds more tightly than other constructs.

Figure 1: The syntax of SHIM

We begin with an informal description of our language and its syntax. We first focus on the exception-free fragment of SHIM (Section 2), then consider exceptions (Section 3). We develop the breadth-first-search example in Section 4. We provide the formal semantics of SHIM with exceptions (Section 5), describe a basic compiler for it (Section 6), discuss related work (Section 7), and conclude (Section 8).

2 The Basic SHIM Language

SHIM, whose syntax is summarized in Figure 1, draws from many familiar sources. Its core is an imperative language with C-style syntax and semantics that includes local variable declarations and procedure calls, but not pointers. Our procedures have both pass-by-value and pass-by-reference parameters (those prefixed with the C++-style &). Our language does not have functions per se, but procedures can return values through pass-by-reference arguments.

In addition to the usual imperative statements, SHIM has four novel ones:

- "s `par` s" for concurrency,
- "`next` V`;`" for communication,
- "`try` s `catch(` E `)` s" to define and handle exceptions, and
- "`throw` E`;`" to raise exceptions.

2.1 Concurrency

The "p `par` q" statement runs p and q concurrently and waits for both p and q to complete their execution before it terminates. In other words, it forks two threads of execution responsible for running p and q and suspends the current thread until the completion of both. As a result, a parent thread never runs at the same time as its subthreads.

By design, *par* is commutative and associative. For instance, "p `par` q `par` r" and "q `par` r `par` p" behave identically. However, because of variable scope, "`{`p `par` q`}` `par` r" may behave differently, just as "`{`p`}` q" and "p q" may be different in C.

The SHIM scheduler is allowed to arbitrarily interleave the execution of concurrently-running threads, provided doing so does not violate inter-thread communication rules. In general, concurrent threads execute *asynchronously*.

To achieve deterministic behavior—behavior independent from arbitrary scheduling choices—we impose restrictions on "shared variables"—the constructs through which concurrent threads may communicate. In a *par* statement, zero or one thread is passed each variable by reference; one or more threads may be passed the same variable by value.

We rely on a simple syntactic rule to choose which thread (if any) gets the by-reference variable: a variable is an *lval* for a thread and passed-by-reference iff it appears to the left of an assignment or is bound to a pass-by-reference argument in a procedure call; a variable is an *rval* for a thread if it only occurs in expressions, after the *next* keyword, or is bound to pass-by-value arguments only in procedure calls. A variable must not be an *lval* for two threads or more in a *par* statement.

For example,

```
f(int &x) {}     // pass-by-reference
g(int x) {}      // pass-by-value
main() {
  int a; a = 0;
  int b; b = 0;
  a = 1; par b = a;   // OK
  a = 1; par a = 2;   // No: a is an lval twice
  f(a); par f(b);     // OK
  f(a); par g(a);     // OK
  g(a); par g(a);     // OK
  f(a); par f(a);     // No: a is an lval twice
}
```

An *lval* is passed by reference to the thread, an *rval* is passed by value. For example,

```
main() {
  int a; a = 3;
  int b; b = 7;
  int c; c = 1;
  {               // lval: a, rval: b, c
    a = a + c;
                  // a is 4, b is 7, c is 1
    a = a + b;
                  // a is 11, b is 7, c is 1
  } par {         // lval: b, rval: a, c
    b = b - c;
                  // a is 3, b is 6, c is 1
    b = b + a;
                  // a is 3, b is 9, c is 1
  }
                  // a is 11, b is 9, c is 1
}
```

Here *c* is passed by value to both threads. Nevertheless, it is correct and more efficient to pass it by reference since neither thread updates *c*. In general, a variable that is an *rval* for all the threads can be safely passed by reference to all of them. Thanks to this optimization, few variables are ever passed by value.

2.2 Communication

The restriction than no variable may be passed by reference to more than one thread at a time makes it impossible for a thread to modify another thread's copy of a variable through a simple assignment. Instead, inter-thread communication is performed by the *next* instruction, which forces threads to synchronize before exchanging values. Depending on how a variable is passed to a thread, *next* either transmits the new global value of a variable or receives the new global value. For example,

```
f(int a) {   // a is a copy of c
  a = 3;
  next a;         // a gets c's value
                  // a = 5
}
g(int &b) {  // b is an alias for c
  b = 5;
  next b;         // synchronize with f
                  // b = 5
}
main() {
  int c; c = 0;
  f(c); par g(c);
}
```

Both *a* and *b* are incarnations of *c*. The *next* instructions assign the current value of *c* to *a* and *b*. Since *c* was passed by value to *f* and by reference to *g*, "`next b`" and "`next a`" behave as a send operation in *g* and a receive operation in *f* respectively, which together result in assigning *b*'s value to *a*.

To make communication deterministic, *next* instructions forces all threads sharing the variable to synchronize. In the previous example, "`next a`" and "`next b`" execute simultaneously.

Such synchronizations may cause deadlocks. For example,

```
main() {
  void a; void b;
  { next a; next b; } par { next b; next a; }
}
```

deadlocks because the branches share *a* and *b* and the first branch is waiting for a synchronization on *a* while second branch is waiting on *b*.

In SHIM, variables with *void* type provide pure synchronization.

A thread is only required to synchronize on a variable it shares. For example,

```
main() {
  void a; void b;
  { next a; next b; } par { next b; }
}
```

does not deadlock because the right branch does not share *a*.

If a thread terminates, it is no longer compelled to participate in a synchronization and therefore does not cause a deadlock. For example,

```
main() {
  void a;
  { next a; next a; } par { next a; }
}
```

does not deadlock. Only the first synchronization on *a* involves both threads.

Pending synchronizations may take place in any order. In

```
main() {
  void a; void b;
  next a; par next b; par next b; par next a;
}
```

the synchronization on *a* may occur before or after the synchronization on *b*.

2.3 Delegation

If a thread divides into subthreads, the parent thread effectively delegates its ownership of a variable to all of its children that use the variable, meaning they are required to participate in any communication on this variable. For example,

```
main() {
  void a; void b;
  { { next a; next b; } par {} } par { next b; a; }
}
```

deadlocks. The rightmost branch knows about *a* and therefore must participate in communication on *a*.

In contrast,

```
main() {
  int a; a = 0;
  int b; b = 0;
  {               // thread 1:   rval: a, b
    {             // thread 1.1: lval a
      next a;
                  // a is 1, b is 0
    } par {       // thread 1.2: lval b
      next b;
                  // a is 0, b is 2
    }
                  // a is 1, b is 2
  } par {         // thread 2:   lval: a, b
    b = 2;
    next b;
    a = 1;
    next a;
  }
}
```

does not deadlock. Thread 2 synchronizes with thread 1.2 first, then with thread 1.1. Although *a* and *b* are passed by value to thread 1, *a* is then passed to 1.1 by reference and *b* to thread 1.2. Consequently, the *next* instructions in threads 1.1 and 1.2 behave as receive operations and return the received values to thread 1.

The pattern "`next a; par next b;`" is a convenient idiom for synchronizing and communicating on *a* and *b* in any order.

If no subthread is responsible for a variable of the parent thread, the parent thread remains responsible for it. For example,

```
main() {
  void a; void b;
  {                       // thread 1:   shares a and b
      next a;             // thread 1.1: shares a
    par
      next b;             // thread 1.2: shares b
  } par {                 // thread 2:   shares a and b
    next a; next a; b;
  }
}
```

deadlocks. After the first synchronization on *a*, thread 1 remains sensitive to *a* even if its now-unique remaining subthread is sensitive to *b* only. Therefore, thread 1 recovers the responsibility for *a* that was temporarily held by one of its subthreads. A synchronization on *a* must involve thread 1, thus the deadlock.

In contrast,

```
main() {
  void a; void b;
  {                       // thread 1: shares a
    next a;
  } par {                 // thread 2: shares b
    next b;
  } par {                 // thread 3: shares a and b
    next a; next a; b;
  }
}
```

does not deadlock. Grouping matters!

In summary, a communication takes place iff all leaf nodes of the tree of threads that share a common variable, i.e., were passed the variable by value or by reference, are ready to execute a *next* instruction for this variable or one of its copies. The tree for each variable evolves dynamically as threads are created and terminate.

2.4 A FIFO Example

The example below is a simple pipeline with feedback consisting of a procedure that increments its input (*f*) and two calls of a buffer procedure (*g*). The pipeline passes around a 1, then a 2, a 3, etc.

```
f(int a, int &b) {
  while (true) {
    b = a + 1;
    next b;    // sends b since b is passed by reference
    next a;    // receives a since a is passed by value
  }
}
g(int b, int &c) {
  while (true) {
    next b;     // receives
    c = b;
    next c;     // sends
  }
}
main() {
  int a; int b; int c;
  a = 0;
  f(a,b); par g(b,c); par g(c,a);
}
```

Starting from the same buffer procedure (*g*), the code below uses recursion and concurrency to implement a FIFO of size *n*.

```
fifo(int i, int &o, int n) {
  int c;
  int m; m = n - 1;
  if (m) {
    g(i, c); par fifo(c, o, m);
  } else {
    g(i, o);
  }
}
```

3 Exceptions in SHIM

The inter-thread communication facility provided by *next* can be used to pass control messages among threads (e.g., "please terminate now"), but doing so is somewhat awkward. SHIM's exception mechanism is layered on the inter-process communication mechanism to preserve determinism while providing powerful sequential control.

Exceptions are scoped, caught, and handled by the *try-catch* construct and raised by the *throw* instruction. Exception declarations may be nested. Exceptions do not carry values.

In sequential code, SHIM's exceptions are classical: the *throw* instruction behaves as a jump to the matching handler, unrolling the stack as necessary. For example,

```
main() {
  int i; i = 0;
  try {
    i = 1;
    throw T;
    i = i * 2;                  // is not executed
  } catch(T) { i = i * 3; }
                                // i = 3
}
```

3.1 Best-Effort Preemption

Concurrent preemption in SHIM follows the principle of best effort. Exceptions may be raised from subthreads, and whether concurrently-running threads in the scope of the exception are affected depends on communication. For example,

```
main() {
  int i; i = 0;
  try {                            // thread 1
    throw T;
  } par {                          // thread 2
    while (true) { i = i + 1; }     // runs forever
  } catch(T) {}
}
```

never terminates. The two threads never communicate and hence never synchronize. Thread 1 thus has no way to interrupt thread 2 at a deterministic point of its execution (e.g., at a deterministic value of *i*). Thread 2 runs forever. Such a pattern will generate a compiler warning.

We say a thread is *poisoned* iff it raises or propagates an exception. If a thread attempts to communicate with a poisoned thread, it gets poisoned and propagates the exception, i.e., it behaves as if it had thrown the same exception. For example,

```
main() {
  int i; i = 0;
  int j; j = 0;
  try {             // thread 1: shares i
    while (i < 5) {
      i = i + 1;
      next i;
    }
    throw T;
  } par {           // thread 2: shares i and j
    while (true) {
      next i;       // is eventually poisoned by thread 1
      j = j + i;
      next j;
    }
  } par {           // thread 3: shares j
    while (true) {
      next j;       // is eventually poisoned by thread 2
    }
  } catch(T) {}
                    // i is 5, j is 15
}
```

terminates. Thread 1 poisons thread 2 that in turn poisons thread 3, even though no variable is shared between threads 1 and 3.

We say a thread is *dying* iff it is in the scope of an exception raised or propagated by one of its subthreads but is still alive because another of its subthreads is still running, being unaffected by the exception. If a thread attempts to communicate with a dying subthread, it gets poisoned as well.

```
main() {
  void i;
  void j;
  void k;
  try {              // thread 1:   shares i, j, and k
    {                // thread 1.1: shares i
      i;
    } par {          // thread 1.2: shares j
      j;
      throw T;
    } par {          // thread 1.3: shares k
      while (true) { // runs forever
        next k;      // synchronize with thread 4
      }
    }
  } par {            // thread 2:   shares i
    next i;          // is poisoned by thread 1
  } par {            // thread 3:   shares j
    next j;          // is poisoned by thread 1.2
  } par {            // thread 4:   shares k
    while (true) {    // runs forever
      next k;        // synchronize with thread 1.3
    }
  } catch(T) {}
}
```

In this example, thread 3 gets poisoned while attempting to communication with thread 1.2. Thread 4 communicates with thread 1.3, which is running fine. Thread 2 attempts to communicate first with thread 1.1 then with thread 1 itself since, upon the completion of thread 1.1, thread 1 becomes responsible for *i*. Thread 1 is dying due to exception *T* in thread 1.2. Therefore, thread 2 gets poisoned. Here again, we observe that thread 1, while dying, never dies since thread 1.3 never returns.

3.2 Scoping

Poison does not flow beyond the scope of its exception. In

```
main() {
  int i; i = 0;
  int j; j = 0;
  {                  // thread 1:   shares i
    try {            // thread 1.1: shares i
      i;
      throw T;
    } par {          // thread 1.2: shares i
      i = i + 1;     // is executed
      next i;        // is poisoned by thread 1.1
      i = i + 1;     // is not executed
    } catch(T) {}
  } par {            // thread 2:   shares i and j
    j = j + 1;       // is executed
    next i;          // executes normally
    j = j + 1;       // is executed
  }
                     // i is 1, j is 2
}
```

exception *T* prevents *i* from being incremented a second time. However, it does not poison thread 2, which is outside the scope of the exception. The *next* instruction in thread 2, which would have to synchronize with the *next* instruction in thread 1.1 if *T* had not been raised, is instead delayed until the completion of the *try-catch* block, at which point it may take place as usual, since thread 2 is the only remaining thread sharing *i*.

3.3 Concurrently-thrown Exceptions

When exceptions are raised in parallel, the outermost exception takes priority and defines the exit point.

```
main() {
  int i; i = 0;
  try {
    try {
      throw T;
    } par {
      throw U;
    } catch(T) {}
    i = i + 1;      // not executed
  } catch(U) {}
                    // i is 0
}
```

When exceptions are raised concurrently at different levels of the thread hierarchy, closer exceptions are dealt with first. For example, in

```
main() {
  int i; i = 0;
  try {             // thread 1:   shares i
    i;
    throw T;
  } par {           // thread 2:   shares i
    try {           // thread 2.1: shares i
      i;
      throw U;
    } par {         // thread 2.2: shares i
      next i;       // is poisoned by thread 2.1
    } catch(U) {}
    i = i + 1;      // is executed
    next i;         // is poisoned by thread 1
    i = i + 1;      // is not executed
  } catch(T) {}
                    // i is 1
}
```

exception U ensures that i is incremented once by poisoning the *next* instruction in thread 2.2, thus preventing exception T to poison it. Exception T only poisons the other *next* instruction so that i is not incremented again.

3.4 The FIFO Revisited

Using an exception, we can terminate the FIFO we described earlier.

```
source(int &a) {
  while (a > 0) {
    a = a - 1;
    next a;     // sends a
  }
  throw T;
}
sink(int b) {
  while (b != 0) {
    next b;     // receives b
  }
  // do something else
}
main() {
  int a; a = 5;
  int b; b = -1;
  int n; n = 3;
  {
    try {
      source(a); par fifo(a, b, n);
    } catch(T) {}
  } par {
    sink(b);
  }
}
```

The *source* sends the values 4, ..., 0 to the three-place FIFO that delivers them to the *sink*. Thanks to poisoning rules, the exception T poisons each one of the three one-place buffers of the FIFO only after it has finished transmitting the five values and becomes receptive again to a sixth value. In other words, the FIFO completes its transmission before terminating. On the other hand, because the sink is not part of the scope of T, it is unaffected by T.

Importantly, if we omit the *throw* instruction in the source, we end up with rather different behavior. Indeed, after the termination of the source, the receiving *next* instruction of the first buffer in the FIFO is no longer compelled to synchronize with anyone and thus behaves as a no-op; the FIFO repeatedly outputs the last transmitted value.

Alternatively, we could terminate the FIFO by raising T in the sink procedure, provided we modify the scope of T accordingly.

In summary, thanks to exceptions, there is no need to hard code a termination condition or an end-of-stream data token in our FIFO.

4 Example: Concurrent Breadth-first Search

In this section, we combine recursion, concurrency, and exceptions to implement two breadth-first-search algorithms in binary trees.

While there are no pointers or objects in the language we have described, we can easily extend SHIM and define a type for binary trees in a Java-like syntax:

```
class Tree {
  int key;
  int value;
  Tree left;
  Tree right;
};
```

Each node associates an integer *value* to an integer *key*.

In general, pointers or objects may introduce aliasing and thus break the determinism of SHIM. However, in the examples below we shall only access binary trees in a read only manner. Therefore, there is no danger of a race. We assume these binary trees are created by sequential code beforehand. We leave the discussion of data structures and pointers in SHIM for future work.

4.1 The depth-first-search mem algorithm

Let us start with a sequential algorithm for deciding membership of a key in a tree.

```
mem(int key, Tree tree) {
  if(tree != null) {
    if(key == tree.key) throw Found;
    mem(key, tree.left);
    mem(key, tree.right);
  }
}
mem2(int key, Tree tree, int &found) {
  found = 0;
  try {
    mem(key, tree);
  } catch(Found) { found = 1; }
}
```

The recursive *mem* procedure reports success through the *Found* exception. The *mem2* procedure produces a Boolean result instead. Overall, this implements a depth-first-search algorithm that gives priority to the left.

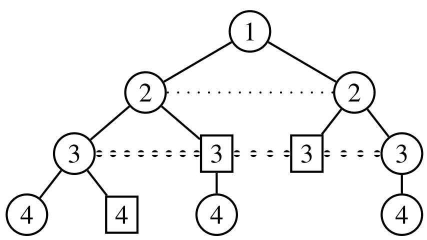
4.2 *The breadth-first-search* mem *algorithm*

To turn the depth-first-search algorithm into a breadth-first-search one in SHIM, we add two lines:

```
mem(int key, Tree tree) {
  if (tree != null) {
    if(key == tree.key) throw Found;
    next key;                          // addition 1
      mem(key, tree.left);
    par                                // addition 2
      mem(key, tree.right);
  }
}
```

The purpose of the second addition is obvious: it turns the sequence of the two recursive calls into their parallel composition.

The first addition is more subtle. Basically, it makes it possible for a branch that finds the *key* to interrupt concurrent branches by forcing them to synchronize. The synchronization pattern is illustrated below: all the recursive calls corresponding to the same depth (same node label) synchronize as shown by the dashed lines.



If the key is found at a square node, then all recursive calls terminate at this (third) level either by raising the exception (square nodes) or by getting poisoned (circular nodes). Nodes at the fourth level and below are not visited.

Incidentally, because the key's value is never modified, the *next* instruction is used here for synchronization purpose only. A dedicated *tick* channel could be used instead:

```
mem(int key, Tree tree, void tick) {
  if (tree != null) {
    if (key == tree.key) throw Found;
    next tick;                    // synchronizes on tick
      mem(key, tree.left, tick);  // shares tick
    par
      mem(key, tree.right, tick); // shares tick
  }
}
```

4.3 *The breadth-first-search* assoc *algorithm*

We now would like to improve the breadth-first-search algorithm to return the value associated with the key. One tentative extension of the previous algorithm is the following:

```
assoc(int key, Tree tree, int &value) {
  if (tree != null) {
    if(key == tree.key) {
      value = tree.value;
      throw Found;
    }
    next key;
      assoc(key, tree.left, value);    // lval: value
    par
      assoc(key, tree.right, value);   // lval: value
  }
}
```

But this code is rejected by the compiler as incorrect: *value* is passed by reference to both parallel recursive calls, which may cause a race. Consider again the example of the previous section. It contains the key in two nodes at level 3. Which value should be returned? The leftmost or the rightmost value? The above piece of code does not specify a deterministic behavior, but fortunately our language enforces determinism and rejects the above program.

Here is a correct algorithm:

```
assoc(int key, Tree tree, int &value) {
  if(tree != null) {
    if(key == tree.key) {
      value = tree.value;
      throw Found;
    }
    next key;
    int tmp;
    try {
      assoc(key, tree.left, value);
    } par {
      try {
        assoc(key, tree.right, tmp);
      } catch(Found) { throw Right; }
    } catch(Right) { value = tmp; throw Found; }
  }
}
assoc2(int key, Tree tree, int &value) {
  value = -1;
  try {
    assoc(key, tree, value);
  } catch(Found) {}
}
```

First, we introduce a temporary variable *tmp* to hold the value returned by the recursive call for the right subtree. An assignment from *tmp* to *value* may only occur in the handler of the *Right* exception, whose scope contains the parallel recursive calls. Hence, the assignment is in sequence after the parallel composition. This is correct.

Second, we use exceptions to implement priorities. There are four cases:

1. Neither branch raises exception *Found*: the procedure terminates normally or runs forever if the tree is infinite.

2. The left branch only raises exception *Found*: the exception kills the right branch and propagates upwards. The variable *value* contains the value returned by the left branch.

3. The right branch only raises exception *Found*: the exception is caught and *Right* is raised. Exception *Right* kills the left branch. The exception handler for *Right* assigns *tmp* to *value*. Exception *Found* is raised and propagates upwards. The variable *value* contains the value returned by the right branch.

4. Both branches raise exception *Found*: the exception propagates upwards, in particular preempting the execution of the handler for the *Right* exception. The variable *value* contains the value returned by the left branch.

In summary, the algorithm returns the value associated with the leftmost node among those nodes that match the key and have the shortest distance to the root of the tree.

Thanks to concurrency, recursion, and exceptions we can implement in SHIM a deterministic breadth-first-search algorithm, and no nondeterministic algorithm can be expressed in SHIM.

5 The Semantics of SHIM

Here, we provide a formal operational semantics of SHIM. We express the execution of a program as a set of rules that specify the possible transitions between program states. For simplicity, we first address concurrency and communication in SHIM (Section 5.1), then consider exceptions (Section 5.2). We conclude with a discussion of determinism (Section 5.3).

5.1 Concurrency and Communication

One challenge here is choosing an appropriate notion of state. We start with an informal discussion of the various components of a state, describe our state encoding, explain the rules of the formal semantics, and demonstrate the semantics by showing how it executes an example. In Section 6, we describe a concrete implementation of these semantics.

In our semantics, we express the state of a program as a pair consisting of a *residue*, which specifies the code remaining to be executed in each thread and the hierarchy of threads; and a *store*, which describes the current memory layout and content. The initial state consists of the body of the *main* procedure and an empty store.

The store maps locations to values. A variable declaration allocates a fresh location in the store and binds it to the name of the variable being declared. An assignment to a variable updates the value of its location in the store.

Since SHIM is concurrent, we define the residue to be a tree of code fragments that encodes the hierarchy of threads. Only the leaves of a residue run concurrently; the execution of a non-leaf node only proceeds when its children have terminated and disappeared from the tree. A *par* statement augments the tree by adding concurrently-running children under the node of the current thread.

Each node in the residual tree maintains a *view*. Each view binds the variable identifiers visible to the thread to locations in the store. This is a single location for a local variable; each parameter is actually a pair of locations: one that holds the current value of the parameter; another that tracks the location of the shared variable location, i.e., the source location of the data copied in a *next* operation.

Although for simplicity our semantics uses a single, global store shared across all threads, SHIM can be implemented in a distributed, message-passing style. Our views capture data locality—a thread may only access data in its view. Information may only flow between concurrently-running threads at *next* instructions that perform a sort of message passing.

5.1.1 Notation

For simplicity, we assume the parameters, local variables of a thread, and procedure names have pairwise distinct identifiers. We only consider well-scoped, well-typed programs. In particular, we assume that all procedures calls are matched by declarations with matching arities.

For a procedure p, $\mathrm{param}(p)$ are the formal parameters of p; $\mathrm{param}(p)_i$ is the ith parameter of p. We write $\mathrm{byref}(p)$ and $\mathrm{byval}(p)$ for the sets of by-reference and by-value parameter indices respectively. Moreover, $\mathrm{body}(p)$ stands for the sequence of statements in p.

Our semantics holds the values of variables in a store. $\Lambda = \{\lambda, \mu, \dots\}$ denotes an infinite set of abstract locations and $\mathscr{V}$ denotes values. A store is a partial function $\sigma : \Lambda \to \mathscr{V}$. $\mathrm{Dom}(\sigma)$ denotes the domain of the store σ, which we require to be finite. By design, uninitialized locations $\lambda \in \mathrm{Dom}(\sigma)$ have value $\perp$. We often use a set-like notation to define the function of a store, e.g., $\{\lambda \mapsto 0, \mu \mapsto \perp\}$ denotes a store σ where $\sigma(\lambda) = 0$ and $\sigma(\mu) = \perp$.

A view can be thought of as a symbol table that maps variable names to locations in the store. Technically, a view $v : V \to \Lambda + (\Lambda \times \Lambda)$ is a partial function from a finite set of variable identifiers to locations (for local variables) or pairs of locations (for parameters). If $v(x) = \lambda$ then we define $v_{\mathrm{loc}}(x) = v_{\mathrm{glb}}(x) = v(x)$, otherwise we define $v(x) = v_{\mathrm{loc}}(x), v_{\mathrm{glb}}(x)$. By design, $v_{\mathrm{loc}}(x)$ points to the current value of name x, whereas $v_{\mathrm{glb}}(x)$ retains the shared variable location associated with name x. We denote by $\mathrm{Glb}(v)$ the image of v_{glb} and by $\mathrm{Def}(v)$ the locations of the local variable identifiers in $\mathrm{Dom}(v)$, i.e., the locations λ such that $\exists x \in \mathrm{Dom}(v) : v(x) = \lambda$. For instance $\{x \mapsto \lambda, y \mapsto \alpha, \beta\}$ denotes a view v such that $v_{\mathrm{loc}}(x) = \lambda$, $v_{\mathrm{loc}}(y) = \alpha$, $v_{\mathrm{glb}}(x) = \lambda$, $v_{\mathrm{glb}}(y) = \beta$, $\mathrm{Dom}(v) = \{x, y\}$, $\mathrm{Glb}(v) = \{\lambda, \beta\}$, $\mathrm{Def}(v) = \{\lambda\}$.

A state—the main object manipulated by the semantics—is a pair r/σ, where σ is a store and r a residue such that all locations appearing in r are in $\mathrm{Dom}(\sigma)$. A residue r is a tree whose nodes are pairs $s^*|v$ that combine a sequence of statements with a view. $\mathbf{0}$ denotes the empty list of statements. We denote by $R \triangleright s^*|v$ compound residues where $s^*|v$ is the root of the tree and R is a non-empty multiset of residues that denotes the branches of the tree. For instance, the state marked with $(*)$ in Figure 4 has root $\mathbf{0}|\{\mathtt{c} \mapsto \lambda\}$ and leaves $\mathtt{f(c);}|\{\mathtt{c} \mapsto \mu, \lambda\}$ and $\mathtt{g(c);}|\{\mathtt{c} \mapsto \lambda, \lambda\}$. Branch ordering is irrelevant. The store is $\{\lambda \mapsto 0, \mu \mapsto 0\}$.

If residue r has a root node with view v we define $\mathrm{Glb}(r) = \mathrm{Glb}(v)$ and $\mathrm{Def}(r) = \mathrm{Def}(v)$.

5.1.2 Formal Semantics

In Figure 2, we formalize the semantics of SHIM without exceptions as a set of deduction rules in a structural operational style [31].

In addition to what we previously described, we add an extra piece of information atop $\triangleright$ symbols, which relates to exception scopes. It can be ignored for the moment; we shall discuss it in Section 5.2;

The rules for *if*, *while* statements and assignments are standard. An auxiliary function $\mathscr{E}$, which we do not define here, computes expression values. It implements a deterministic Java-like evaluation strategy. It resolves local variable and parameter names to values using function $\sigma \circ v_{\text{loc}}$.

At a local variable declaration, we bind the name of the variable to a new location whose value starts at $\perp$. As usual, the semantics is defined up to alpha-renaming of locations.

The *block* rule uses a trick to correctly scope additional variable declarations: it forks a single child node that contains the body of the block and copies the current view.

The *context* rule derives a step for a node from a step of one of its children. $\uplus$ denotes the union of multisets.

The *join* and *return* rules take care of the completion of function calls, blocks, and parallel branches. Rule *join* handles one branch at a time so that a branch releases its channels immediately upon termination. When the last branch terminates, the execution of the parent thread is resumed thanks to rule *return*.

The *next*, *gather*, and *sync* rules handle synchronization. The *next* rule specifies that a leaf code fragment starting with a *next* instruction for variable x may synchronize on location $v_{\text{glb}}(x)$ and expects $v_{\text{loc}}(x)$ to be updated to reflect the current value of $v_{\text{glb}}(x)$. Thanks to the *gather* rule, an inner node may synchronize on location λ provided all its child nodes that know about λ agree on such a synchronization. Rule *sync* proceeds with the synchronization at the node where λ was initially allocated: all local copies of the shared variable are atomically updated.

The *call* rule handles procedure calls. It allocates new locations for by-value parameters, which are initialized with the values of the actual parameters. It also expands the body of the callee and creates a view for it. This view binds the formal parameters of the procedure to their actual values.

The *par* rule iterates the *branch* rule to handle parallel compositions. The *branch* rule resembles the *call* rule except that it relies on the Lval and Rval sets obtained by static analysis to decide which variables are passed by reference to the thread. Importantly, variables that do not occur in the thread are not part of the view of the thread.

Observe the *next*, *gather*, and *call* rules are only meant to produce intermediate results that are eventually used by rules *sync* and *par* to prove regular transitions.

5.1.3 Example

Figure 4 shows one possible execution of the example in Section 2.2. We decorate each transition with a skeleton of its proof tree. Starting from the body of the *main* procedure, the execution first proceeds with the local variable declaration, the assignment and the concurrent procedure calls that fork two parallel threads. Since parallel branches may execute asynchronously, several transitions may in general be taken from a given program state. In particular, the two transitions for the two local assignments may occur in any order. After the two assignments, a synchronization takes place. It updates the value at location ν with the value at location λ. Finally, both procedures and both branches return and the program terminates.

5.2 Exceptions

In Figure 3, we provide additional rules to handle exceptions in SHIM. Combined with the rules of Figure 2, they form the operational semantics of our language.

To keep track of exceptions scopes in the semantics, we augment the residual structure we described in Section 5.1. From now on, we write

$$R \overset{m}{\triangleright} s^*|v,$$

where m is either an exception identifier e if the $\triangleright$ results from a *try-catch* construct for exception e and 0 otherwise (*block*, *call*, and *par* rules).

We also introduce the placeholder instruction *handler* to denote pending handlers in the residual tree. The *handler* instruction does not appear in the SHIM language itself.

The *try* rule forks a new child node for the body p of the *try-catch* construct, decorates the $\triangleright$ with the exception identifier e, and insert the handler q. The view of p is obtained by copying the current view.

Rules *throw*, *throw2*, and *throw3* handle *throw* statements. First, rule *throw* replaces the body of the thread with a special **X** statement that marks it as a poisoned threads. Second, rule *throw2* marks enclosing threads with the same **X** to indicate they are dying. Finally, rule *throw3* stops the poisoning at the boundary of the exception scope.

Rules *exit* and *handler* handle the completion of poisoned threads. When all subthreads of a thread are poisoned, they are deleted. The corresponding exception handler is executed, if any, if the poisoning ends at the parent thread.

Rule *skip-handler* specifies that handlers must not be executed if the exception was not raised, i.e., following a *return* transition.

Rules *exception*, *exception2*, and *exception3* decide whether a communication attempt with a thread r on a shared variable with global location λ poisons the thread attempting the communication: $\lambda \in \text{Exc}(r)$.

Rules *next-fail*, *gather-fail*, *sync-fail* propagate poison from poisoned or dying threads to threads that attempt to communicate with them. Rule *next-fail* and *gather-fail* are duals of rules *next* and *gather* except they prepare for a communication failure, that is to say poison propagation, rather than a success. Rule *sync-fail* matches a communication attempt with a poisoned or dying thread and proceeds with the poisoning of the thread attempting the communication.

5.3 Determinism

We claim that our semantics are deterministic in Kahn's sense: *computations* and *communications* are the same for all *fair executions*. Intuitively, this follows from our processes following the Kahn principle: communication takes place exclusively through channels, is blocking, and each thread of control can block on at most one communication at once and cannot retreat from an attempt to communicate.

More precisely, our transition system is locally confluent. Informally, whenever several transitions are possible from

$$\frac{\mathscr{E}(e,\sigma\circ v_{\mathrm{loc}})\neq 0}{\texttt{if}\ (e)\ p\ \texttt{else}\ q\ s^*|v\,/\,\sigma \longrightarrow p\ s^*|v\,/\,\sigma}\ \text{(if)} \qquad \frac{\mathscr{E}(e,\sigma\circ v_{\mathrm{loc}}) = 0}{\texttt{if}\ (e)\ p\ \texttt{else}\ q\ s^*|v\,/\,\sigma \longrightarrow q\ s^*|v\,/\,\sigma}\ \text{(else)}$$

$$\frac{\mathscr{E}(e,\sigma\circ v_{\mathrm{loc}})\neq 0}{\texttt{while}\ (e)\ p\ s^*|v\,/\,\sigma \longrightarrow p\ \texttt{while}\ (e)\ p\ s^*|v\,/\,\sigma}\ \text{(while)} \qquad \frac{\mathscr{E}(e,\sigma\circ v_{\mathrm{loc}}) = 0}{\texttt{while}\ (e)\ p\ s^*|v\,/\,\sigma \longrightarrow s^*|v\,/\,\sigma}\ \text{(wend)}$$

$$\frac{\mathit{true}}{x\texttt{=}e\texttt{;}\,s^*|v\,/\,\sigma \longrightarrow s^*|v\,/\,\sigma\{v_{\mathrm{loc}}(x)\mapsto\mathscr{E}(e,\sigma\circ v_{\mathrm{loc}})\}}\ \text{(assign)} \qquad \frac{R\neq\emptyset}{\{\mathbf{0}|v'\}\uplus R\overset{m}{\triangleright} s^*|v\,/\,\sigma \longrightarrow R\overset{m}{\triangleright} s^*|v\,/\,\sigma}\ \text{(join)}$$

$$\frac{\lambda\notin\mathrm{Dom}(\sigma)}{t\ x\texttt{;}\,s^*|v\,/\,\sigma \longrightarrow s^*|v\{x\mapsto\lambda\}\,/\,\sigma\{\lambda\mapsto\bot\}}\ \text{(declare)} \qquad \frac{\mathit{true}}{\{\mathbf{0}|v'\}\overset{m}{\triangleright} s^*|v\,/\,\sigma \longrightarrow s^*|v\,/\,\sigma}\ \text{(return)}$$

$$\frac{r\,/\,\sigma \longrightarrow r'\,/\,\sigma'}{\{r\}\uplus R\overset{m}{\triangleright} s^*|v\,/\,\sigma \longrightarrow \{r'\}\uplus R\overset{m}{\triangleright} s^*|v\,/\,\sigma'}\ \text{(context)} \qquad \frac{\begin{array}{ll} v' : \mathrm{Dom}(v) \to \Lambda\times\Lambda & \\ \quad x & \mapsto v_{\mathrm{loc}}(x), v_{\mathrm{glb}}(x)\end{array}}{\{p^*\}\ s^*|v\,/\,\sigma \longrightarrow \{p^*|v'\}\overset{0}{\triangleright} s^*|v\,/\,\sigma}\ \text{(block)}$$

$$\frac{I\neq\emptyset \qquad \forall i\in I : r_i \xrightarrow[M_i]{\lambda} r'_i \qquad \forall j\in J : \lambda\notin\mathrm{Glb}(r_j)}{\{r_i\}_{i\in I}\uplus\{r_j\}_{j\in J}\overset{m}{\triangleright} s^*|v \xrightarrow[\bigcup_{i\in I}M_i]{\lambda} \{r'_i\}_{i\in I}\uplus\{r_j\}_{j\in J}\overset{m}{\triangleright} s^*|v}\ \text{(gather)} \qquad \frac{\mathit{true}}{\texttt{next}\ x\texttt{;}\,s^*|v \xrightarrow[\{v_{\mathrm{loc}}(x)\}]{v_{\mathrm{glb}}(x)} s^*|v}\ \text{(next)}$$

$$\frac{p_0|v\,/\,\sigma \longmapsto r_0\,/\,\sigma_0 \qquad \ldots \qquad p_n|v\,/\,\sigma_{n-1} \longmapsto r_n\,/\,\sigma_n}{p_0\ \texttt{par}\ \ldots\ \texttt{par}\ p_n\ s^*|v\,/\,\sigma \longrightarrow \{r_0,\ldots,r_n\}\overset{0}{\triangleright} s^*|v\,/\,\sigma_n}\ \text{(par)} \qquad \frac{r\xrightarrow[M]{\lambda} r' \qquad \lambda\in\mathrm{Def}(r)}{r\,/\,\sigma \longrightarrow r'\,/\,\sigma\{\mu\mapsto\sigma(\lambda)\}_{\mu\in M}}\ \text{(sync)}$$

$$\frac{\begin{array}{l}\forall x\in\mathrm{Rval}(p) : \lambda_x\notin\mathrm{Dom}(\sigma)\\ \forall x,y\in\mathrm{Rval}(p) : x\neq y \Rightarrow \lambda_x\neq\lambda_y\end{array} \qquad \begin{array}{lll} v' : \mathrm{Lval}(p)\cup\mathrm{Rval}(p) \to \Lambda\times\Lambda & & \\ \quad x & \mapsto \lambda_x, v_{\mathrm{glb}}(x) & \forall x\in\mathrm{Rval}(p)\\ \quad x & \mapsto v_{\mathrm{loc}}(x), v_{\mathrm{glb}}(x) & \forall x\in\mathrm{Lval}(p)\end{array}}{p|v\,/\,\sigma \longmapsto p|v'\,/\,\sigma\{\lambda_x\mapsto\sigma\circ v_{\mathrm{loc}}(x)\}_{x\in\mathrm{Rval}(p)}}\ \text{(branch)}$$

$$\frac{\begin{array}{l}\forall i\in\mathrm{byval}(p) : \lambda_i\notin\mathrm{Dom}(\sigma)\\ \forall i,j\in\mathrm{byval}(p) : i\neq j \Rightarrow \lambda_i\neq\lambda_j\end{array} \qquad \begin{array}{lll} v' : \mathrm{param}(p) & \to \Lambda\times\Lambda & \\ \quad \mathrm{param}(p)_i & \mapsto \lambda_i, v_{\mathrm{glb}}(a_i) & \forall i\in\mathrm{byval}(p)\\ \quad \mathrm{param}(p)_i & \mapsto v_{\mathrm{loc}}(a_i), v_{\mathrm{glb}}(a_i) & \forall i\in\mathrm{byref}(p)\end{array}}{p(a_0,\ldots,a_n)\texttt{;}\,s^*|v\,/\,\sigma \longrightarrow \{\mathrm{body}(p)|v'\}\overset{0}{\triangleright} s^*|v\,/\,\sigma\{\lambda_i\mapsto\sigma\circ v_{\mathrm{loc}}(a_i)\}_{i\in\mathrm{byval}(p)}}\ \text{(call)}$$

Figure 2: The semantics of the exception-free fragment of SHIM.

a given program state, they commute: they may be applied sequentially in any order, all permutations resulting in the same final state. However, due to the size of the semantics we have not completed a formal proof of this result. In fact, due to the dynamic nature of communication channels and units of execution in our model, even a precise theorem is beyond the scope of this paper.

6 A Basic Implementation

In this section, we present a basic implementation of SHIM and some details of its compiler, which generates single-threaded C code. Not surprisingly, implementing concurrency, the *next* statement, and the *try-throw-catch* construct are the main challenges. Our compiler, while preliminary, is able to run all the examples in this paper except those that manipulate non-integer data.

Of course, in the long run, we want to provide compilers for SHIM able to exploit concurrency. In particular, we would like to consider multicore code generation. But for the time being, a single-threaded code generator is sufficient to experiment with our programming model.

Our implementation of SHIM is similar to the basic software translation we presented elsewhere [16]: each procedure is translated into a single C function that uses a state variable and a leading *switch* statement to permit the function to block and resume at communication points. A central scheduler executes ready-to-run functions in a nondeterministic order that does not affect the overall system behavior in accordance with the SHIM semantics.

SHIM's recursion means a program may require unbounded resources, so our compiler produces code that dynamically allocates memory for procedure activation records. Furthermore, because of the concurrency, such activation records cannot simply be stored on a stack (while procedures are properly nested in SHIM, a group of concurrent procedures may call other procedures or terminate in an

$$\frac{\mathit{true}}{\{\mathbf{X}|v_0,\ldots,\mathbf{X}|v_n\} \overset{m}{\triangleright} \mathbf{X}|v \big/ \sigma \longrightarrow \mathbf{X}|v \big/ \sigma} \quad \text{(exit)}$$

$$\frac{\mathit{true}}{\{\mathbf{X}|v_0,\ldots,\mathbf{X}|v_n\} \overset{m}{\triangleright} \texttt{handler}\ q\ s^*|v \big/ \sigma \longrightarrow q\ s^*|v \big/ \sigma} \quad \text{(handler)}$$

$$\frac{\mathit{true}}{\texttt{handler}\ q\ s^*|v \big/ \sigma \longrightarrow s^*|v \big/ \sigma} \quad \text{(skip-handler)}$$

$$\frac{\begin{array}{ll} v' : \mathrm{Dom}(v) \to \Lambda \times \Lambda & \\ \quad x \qquad\qquad \mapsto v_{\mathrm{loc}}(x), v_{\mathrm{glb}}(x) & \end{array}}{\texttt{try}\ p\ \texttt{catch(}e\texttt{)}\ q\ s^*|v \big/ \sigma \longrightarrow \{p|v'\} \overset{e}{\triangleright} \texttt{handler}\ q\ s^*|v \big/ \sigma} \quad \text{(try)}$$

$$\frac{\mathit{true}}{\texttt{next}\ x\texttt{;}s^*|v \xrightarrow[\textbf{fail}]{v_{\mathrm{glb}}(x)} \mathbf{X}|v} \quad \text{(next-fail)}$$

$$\frac{\mathit{true}}{\texttt{throw}\ e\texttt{;}s^*|v \xrightarrow{e} \mathbf{X}|v} \quad \text{(throw)}$$

$$\frac{r \xrightarrow{e} r' \qquad m \neq e}{\{r\} \uplus R \overset{m}{\triangleright} s^*|v \xrightarrow{e} \{r'\} \uplus R \overset{m}{\triangleright} \mathbf{X}|v} \quad \text{(throw2)}$$

$$\frac{r \xrightarrow{e} r'}{\{r\} \uplus R \overset{e}{\triangleright} s^*|v \big/ \sigma \longrightarrow \{r'\} \uplus R \overset{e}{\triangleright} s^*|v \big/ \sigma} \quad \text{(throw3)}$$

$$\frac{\lambda \in \mathrm{Glb}(v)}{\lambda \in \mathrm{Exc}(\mathbf{X}|v)} \quad \text{(exception)}$$

$$\frac{\lambda \in \mathrm{Glb}(v) \setminus \mathrm{Glb}(R)}{\lambda \in \mathrm{Exc}(R \overset{m}{\triangleright} \mathbf{X}|v)} \quad \text{(exception2)}$$

$$\frac{\lambda \in \mathrm{Glb}(v) \cap \mathrm{Exc}(r)}{\lambda \in \mathrm{Exc}(\{r\} \uplus R \overset{m}{\triangleright} \mathbf{X}|v)} \quad \text{(exception3)}$$

$$\frac{I \neq \emptyset \qquad \forall i \in I : r_i \xrightarrow[\textbf{fail}]{\lambda} r'_i \qquad \forall j \in J : \lambda \notin \mathrm{Glb}(r_j)}{\{r_i\}_{i\in I} \uplus \{r_j\}_{j\in J} \overset{m}{\triangleright} s^*|v \xrightarrow[\textbf{fail}]{\lambda} \{r'_i\}_{i\in I} \uplus \{r_j\}_{j\in J} \overset{m}{\triangleright} \mathbf{X}|v} \quad \text{(gather-fail)}$$

$$\frac{r \xrightarrow[\textbf{fail}]{\lambda} r' \qquad \lambda \in \mathrm{Exc}(r_\lambda)}{\{r\} \uplus \{r_\lambda\} \uplus \{r_i\}_{i\in I} \overset{m}{\triangleright} s^*|v \big/ \sigma \longrightarrow \{r'\} \uplus \{r_\lambda\} \uplus \{r_i\}_{i\in I} \overset{m}{\triangleright} s^*|v \big/ \sigma} \quad \text{(sync-fail)}$$

Figure 3: The semantics of exceptions in SHIM.

arbitrary order), so we choose to store them on the heap (i.e., by calling *malloc* and *free*).

The problem of activation record allocation in the presence of concurrency (e.g., through coroutines) is well-understood and the solution is sometimes referred to as a "cactus stack" after their tree-like structure and/or prickly difficulty. For example, the Icon programming language [21] places restrictions on its coroutine-like generators to avoid cactus stacks in most cases (in certain cases, however, it copies activation records to the top of the stack), but simply puts them on the heap in general. Gupta and Soffa [22] suggest that heap-managed activation records are usually as efficient as other, more complicated schemes, so that is the approach we have taken.

6.1 Dismantling

To maintain the invariant that each procedure invocation has exactly one program counter, we dismantle *par* constructs into parallel procedure calls. To simplify the exception-handling machinery, we also dismantle *try-catch* constructs into procedure calls. Both of these simplify the runtime system at the probable expense of efficiency, which was not our objective in developing this implementation.

Our dismantler follows the simple syntactic rules described in Section 2.1 to determine which variables are passed by reference and value to the procedure that implement branches of a *par* construct. Figure 5 shows an example: the three parallel branches of the *par* have become procedures *main_1–main_3*, the body of the *try* has become *main_4*, and the exception *T* has been replaced with the number 0.

```
main() {
  int a; int b; int c;
  a = 1; b = 10; c = 100;
  {
    a = a + 1;
  } par {
    b = a + c;
  } par {
    c = b + 2;
  }
  try {
    a = a + 5;
    throw T;
  } catch (T) {
    b = b + 1;
  }

  // a=7, b=102, c=12
}
```

```
main_1(int &a) { a = a + 1; }
main_2(int &b, int a, int c) {
  b = a + c;
}
main_3(int &c, int b) {
  c = b + 2;
}
main_4(int &a) { throw 0; }
main() {
  int a; int b; int c;
  a = 1; b = 10; c = 100;
  main_1(a); par
  main_2(b, a, c); par
  main_3(c, b);
  main_4(a); catch (0) {
    b = b + 1;
  }
}
```

Figure 5: A SHIM program and the result of dismantling

6.2 Data Types

Figure 6 shows the two fundamental datatypes used by the runtime system. Every procedure's activation record begins with the same fields as *struct ar*, which starts with three pointers that doubly-link the tree of activation records. This tree is traversed when procedures communicate, terminate, or throw exceptions.

$$\texttt{int c; c=0; f(c); par g(c);}|\emptyset\big/\emptyset$$

$$\xrightarrow{\text{declare}} \texttt{c=0; f(c); par g(c);}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto\bot\}$$

$$\xrightarrow{\text{assign}} \texttt{f(c); par g(c);}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 0\}$$

$$\xrightarrow{\frac{\text{branch}\quad\text{branch}}{\text{par}}} \Big\{\texttt{f(c);}|\{\texttt{c}\mapsto\mu,\lambda\},\quad \texttt{g(c);}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 0,\mu\mapsto 0\} \qquad (*)$$

$$\xrightarrow{\frac{\text{call}}{\text{context}}} \Big\{\{\texttt{a=3;next a;}|\{\texttt{a}\mapsto\nu,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\mu,\lambda\},\quad \texttt{g(c);}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 0,\mu\mapsto 0,\nu\mapsto 0\}$$

$$\xrightarrow{\frac{\text{call}}{\text{context}}} \Big\{\{\texttt{a=3;next a;}|\{\texttt{a}\mapsto\nu,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\mu,\lambda\},\{\texttt{b=5;next b;}|\{\texttt{b}\mapsto\lambda,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 0,\mu\mapsto 0,\nu\mapsto 0\}$$

$$\xrightarrow{\frac{\frac{\text{assign}}{\text{context}}}{\text{context}}} \Big\{\{\texttt{next a;}|\{\texttt{a}\mapsto\nu,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\mu,\lambda\},\{\texttt{b=5;next b;}|\{\texttt{b}\mapsto\lambda,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 0,\mu\mapsto 0,\nu\mapsto 3\}$$

$$\xrightarrow{\frac{\frac{\text{assign}}{\text{context}}}{\text{context}}} \Big\{\{\texttt{next a;}|\{\texttt{a}\mapsto\nu,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\mu,\lambda\},\quad \{\texttt{next b;}|\{\texttt{b}\mapsto\lambda,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 5,\mu\mapsto 0,\nu\mapsto 3\}$$

$$\xrightarrow{\frac{\frac{\text{next}}{\text{gather}}\ \frac{\text{next}}{\text{gather}}}{\frac{\text{gather}}{\text{sync}}}} \Big\{\{\mathbf{0}|\{\texttt{a}\mapsto\nu,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\mu,\lambda\},\quad \{\mathbf{0}|\{\texttt{b}\mapsto\lambda,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 5,\mu\mapsto 0,\nu\mapsto 5\}$$

$$\xrightarrow{\frac{\text{return}}{\text{context}}} \Big\{\mathbf{0}|\{\texttt{c}\mapsto\mu,\lambda\},\quad \{\mathbf{0}|\{\texttt{b}\mapsto\lambda,\lambda\}\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 5,\mu\mapsto 0,\nu\mapsto 5\}$$

$$\xrightarrow{\frac{\text{return}}{\text{context}}} \Big\{\mathbf{0}|\{\texttt{c}\mapsto\mu,\lambda\},\quad \mathbf{0}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 5,\mu\mapsto 0,\nu\mapsto 5\}$$

$$\xrightarrow{\text{join}} \Big\{\mathbf{0}|\{\texttt{c}\mapsto\lambda,\lambda\}\Big\}\rhd\mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 5,\mu\mapsto 0,\nu\mapsto 5\}$$

$$\xrightarrow{\text{return}} \mathbf{0}|\{\texttt{c}\mapsto\lambda\}\big/\{\lambda\mapsto 5,\mu\mapsto 0,\nu\mapsto 5\}$$

The third and fourth transition are reproduced below (the store is omitted):

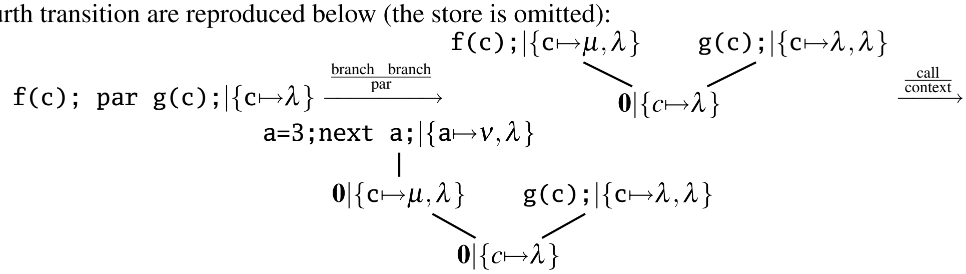


Figure 4: An example of execution.

```
struct channel {
  int *local;
  int caller_channel_index;
  union {
    int intv;
    int *intp;
  } val;
};

struct ar {
  struct ar *caller;
  struct ar *first_callee;
  struct ar *next_sibling;
  void (*fp)(struct ar *);
  int state;
  int exception_index;
  int handler_state;
  int num_channels;
  int blocked_channel;
  struct channel channels[1];
};
```

Figure 6: SHIM runtime data structures for channels and procedure activation records.

The *fp* member is a pointer to the C function implementing the corresponding SHIM function that uses this activation record. Note that these functions take a pointer to a *struct ar*, which is then immediately typecast to a pointer to the actual activation record type for the procedure.

The *state* field holds the control state of the function between invocations. When a function is first called, *state* is zero. When a function blocks by executing a *next* or calls a group of functions, it updates *state* with a small integer to indicate where it should be re-activated. Thus, the *state* variable functions like a return address.

The *exception_index* field indicates the number of the exception, if any, that is handled when control returns to this function. Thus it may change depending on what function it has called. Our compiler dismantles *try* statements into procedure calls with handlers and generates code that unrolls the stack looking for such a matching handler when an exception is thrown. The *handler_state* field indicates the control state the function should enter when handling an exception. It, too, may change while the function is running.

The *num_channels* field indicates the length of the *channels* array. The *blocked_channel* field is normally -1 but

```
struct main_act {                   union main_callsites {
  struct ar (*caller);                struct {
  struct ar (*first_callee);           struct main_1_act f0;
  struct ar (*next_sibling);           struct main_2_act f1;
  void (*fp)(struct ar *);             struct main_3_act f2;
  int state;                          } cs0;
  int exception_index;                struct {
  int handler_state;                   struct main_4_act f0;
  int num_channels;                   } cs1;
  int blocked_channel;              };
  struct channel channels[3];
  union main_callsites *callsites;
};
```

Figure 7: The data types for the *main* procedure in Figure 5

takes on the index of the channel that the procedure is synchronizing on when it is blocked.

The length of the *channels* array in *struct ar* is actually a trick: its length in a real activation record is equal to the number of parameters passed to the procedure plus its local variables. Because C does not check array bounds, our code instead uses the *num_channels* field to ensure we do not overwrite the end of this array.

Each entry in the *channels* array is a *struct channel* that consists of the address of the variable for the channel (the *local* field, which only supports integer variables), the index of the channel in the caller that was passed (-1 for local variables), and a union that holds either an integer (for pass-by-value parameters and local variables) or an integer pointer (for pass-by-reference parameters) that represents the actual value of a variable. It is this field that appears as rvalues and lvalues in SHIM expressions.

The *local* field points to the memory location that is updated when an inter-procedure communication takes place (as in the semantics). For pass-by-value arguments, *local* points into the activation record for the procedure itself, but for pass-by-reference arguments, it points into the activation record where the pass-by-reference variable was defined (i.e., its topmost scope).

Figure 7 shows the data types for the activation record for the *main* procedure of Figure 5. As mentioned above, the fields in this activation record parallel those in *struct ar* (Figure 6) so a pointer to a *struct main_act* can be safely cast to a *struct ar*, but includes a properly-sized *channels* array (three for the two arguments *a* and *b* and the local variable *c*) and a pointer to a *union* that holds activation records for each procedure at every call site (*union main_callsites*).

When a function starts, it calls *malloc()* once to allocate storage for its callsites and *free()* once to free this storage when the function terminates. It is possible that this space is never used because the function never calls any others; optimizing this is future work.

6.3 The Central Scheduler

The central scheduler is straightforward: it simply takes a pointer to an activation record off a stack of runnable processes and calls the function whose pointer is the *fp* pointer within the activation record. The C code looks like

```
  ar->exception_index = 0;
  ar->handler_state = 3; /* handler at case 3: */
  ar->first_callee =
    (struct ar *) &(ar->callsites->cs1.f0);
  ar->callsites->cs1.f0 = (struct main_4_act) {
      (struct ar *) ar, /* caller */
      0,                /* first_callee */
      0,                /* next_sibling */
      main_4_func,      /* fp */
      0,                /* state */
      -1,               /* exception_index */
      0,                /* handler_state */
      1,                /* num_channels */
      -1,               /* blocked_channel */
      { { &(ar->channels[0].val.intv),
          0,
          { .intp=&(ar->channels[0].val.intv) } } } };
  *(++sp) = (struct ar *) &(ar->callsites->cs1.f0);
  ar->state = 4;
  return;
case 3: /* catch 0 */
  ar->channels[1].val.intv =      /* b = b + 1 */
    ar->channels[1].val.intv + 1;
case 4: /* normal termination */
```

Figure 8: Generated code at the *main_4* call site in Figure 5.

```
struct ar *stack[128];
struct ar **sp;

while (sp > stack) {
   --sp;
   ((*(sp))->fp)(*(sp));
}
```

Here, 128 is an arbitrary limit on the number of processes that can be blocked simultaneously, although it is not a limit on the recursion depth—this is not the stack of activation records. This limit could be raised or easily made unbounded.

6.4 Calling Concurrent Procedures

Because the activation record tree may be read before a newly-called procedure has started running, our compiler generates code that initializes the activation records for called functions in the caller, rather than having a function initialize its own activation record.

To simplify the generated code, we generate code that uses the ISO C99 compound literal extensions, which allow us to elegantly state the desired contents of called functions' activation records. Figure 8 shows the code we generate for the `main_4` callsite in Figure 7.

The code in Figure 8 first initializes the *exception_index* and *handler_state* fields in *main*'s activation record since this callsite (created by dismantling a *try-catch*) handles exception 0 (the unique number assigned by the dismantler to T).

Next, it points the first callee field in *main*'s activation record to an activation record for *main_4*, the procedure being called (the *ar* variable points to the activation record for *main*, a *struct main_act*). Then it fills in the activation record in the *cs1* field of *union main_callsites*, pushes its address onto the stack of runnable functions, sets the state of the *main* procedure to 4 (so it will return to the *case 4* label when it terminates normally), and finally passes control back to the global scheduler. If *main_4* terminates normally, control is passed back to the *case 4* label. However, if *main_4* throws the T exception, control is passed to the *case 3* label.

The assignment statement that initializes the activation record fills in a *struct main4_act* using the ISO C99 compound literal syntax. Much like an initialization after a variable declaration, this lists a value for each field. For example, the first field, *caller*, is initialized to the value of *ar*, the activation record for *main*. The *first_callee* field is set to zero since *main_4* has not yet called other procedures. The *next_sibling* field is set to 0 because this callsite is not invoking procedures in parallel.

6.5 Next, Throw, and Termination

The translation of the *next* statement is deceptively simple. If *a* is channel number 1 in the current procedure (e.g., its second parameter) *next(a)* is translated into the C fragment

```
  ar->state = 2;
  next(ar, 1);
  return;
case 2:
```

This prepares the function to resume at the *case 2* label when it resumes (*next* effectively blocks unless all the procedures that participate in the synchronization of the given channel are also trying to synchronize on that channel), attempts a communication action (the call to the *next* function), and passes control back to the central scheduler.

Similarly, if 3 is the encoding for the exception T, *throw T* is translated into

```
  ar->state = 2;
  throw(ar, 3);
  return;
case 2:
```

Finally, a call to the internal function *terminate* is placed at the end of the code for each function to clean up and propagate the effects of its termination. For example, a function may indirectly unblock or poison other functions when it terminates; the *terminate* function is responsible for this.

Broadly, *next* checks if all threads connected to the given channel are ready to communicate on the channel and performs the communication if they are, *throw* walks up the stack to find where the given exception is caught and poisons and terminates what threads it can, and *terminate* removes the thread and tries to unblock the threads that were blocked on the thread.

Figures 9 and 10 show the pseudocode for the runtime system. It is complicated because we are simulating what is fundamentally a concurrent semantics on a sequential machine; an implementation on a parallel machine would be simpler. The three main entry points are *next*, *throw*, and *terminate*; the *evaluate-channel* procedure is the main workhorse underlying them; *poison-subtree* propagates exceptions; *collect-poisoned* cleans up after threads are poisoned and runs *catch* clauses; *channel-poisoned* and *channel-ready* are helper functions that test whether a particular channel has been poisoned or is ready to communicate; and finally *transmit* moves data among ready-to-communicate threads.

The *next* procedure (lines 1–3) marks the thread as blocked (line 2) and calls *evaluate-channel* to attempt the actual communication. Successful communication, which is detected and acted upon in line 31, is the only thing that can unblock a thread, which is done in line 69.

The *throw* procedure (lines 4–9) walks up the call stack and marks every thread as poisoned (line 7) until it finds a function that handles the given exception. Then it calls *collect-poisoned* on the thread in an attempt to propagate the poison and terminate poisoned threads.

The *terminate* procedure (lines 10–17) removes the thread from its parent's list of threads and then either resumes the parent if it was the last surviving child (line 13—this corresponds to a normal function return), or propagates the effects of its termination to all the channels that know about it and possibly to any poisoned siblings that were waiting for it to terminate (line 15terminate-evaluate2).

The *evaluate-channel* procedure (lines 18–33) is the main workhorse, responsible for attempting to communicate on a channel. The are two main cases: the channel has been poisoned by a child of *a* (lines 20–25), or the channel is ready to communicate because all of the children at the top of the tree that know about *c* are blocked on it (lines 26–31).

The first case is when the channel has been poisoned at this level. Unlike communication, which requires every thread that knows about a channel to participate in a communication, poisoning can take place even when threads that know about *c* in other parts of the tree are not yet willing to communicate on *c*. This is because while communication is tied to the scope of the channel, poisoning is tied to the scope of the exception, which is accounted for in the "walk up the tree" logic in the *throw* procedure.

The rules for poisoning are as follows: a channel is poisoned if there is a poisoned path along threads that know about the channel to a leaf. The *channel-poisoned* function (lines 56–60, called in line 21) tests this by trying to walk down the tree along such a path. If a channel is poisoned by a subtree, the poison is spread to every one of its sibling subtrees that is ready to communicate on the channel (if they are not ready to communicate, they might terminate normally before the poison can take hold). The predicate is tested in line 23 and the subtrees are poisoned by the *poison-subtree* procedure (line 34poison-subtree2, called in line 24).

The *poison-subtree* procedure (lines 34–42) walks down a tree to every thread that knows about the channel *c*, poisoning them all. When it reaches a leaf, it propagates the effect of the poison to every channel it knows about and attempts to clean up completely-poisoned threads (lines 40–42).

If no child of a subtree is poisoned, control reaches line 26 in *evaluate-channel*. If we are at the top of the tree for channel *c* (tested in line 26), then we call *channel-ready* (lines 61–65, called in line 28) on each of the children that know about *c*. If they are all ready and there is at least one,

```
1: procedure next(activation record a, channel c)
2:   Mark a as blocked on c
3:   evaluate-channel(a, c)
4: procedure throw(activation record a, exception e)
5:   p ← a
6:   while p does not handle e do                 poison up to handler
7:     mark p as poisoned
8:     p ← p's caller
9:   collect-poisoned(a)
10: procedure terminate(activation record a)
11:   remove a from caller's children
12:   if a's caller now has no children then
13:     schedule a's caller                 all parallel calls terminated
14:   else
15:     for each channel c known to a do
16:       evaluate-channel(a, c)                         unblock peers
17:     collect-poisoned(a)                     any poisoned siblings?
18: procedure evaluate-channel(activation record a, channel c)
19:   while c is a channel in a do
20:     for each child s of a that knows about c do
21:       if channel-poisoned(s, c) then
22:         for each child s of a that knows about c do
23:           if channel-ready(s, c) then
24:             poison-subtree(s, c)
25:         return
26:     if c is not defined in a's caller then             top of the tree
27:       for each child s of a that knows about c do
28:         if not channel-ready(s, c) then
29:           return                not ready to communicate: block
30:       if some child knows about c then
31:         transmit(value of c in a, a, c)
32:     c ← channel for c in caller of a, if it exists
33:     a ← caller of a
```

Figure 9: The runtime algorithms (1/2)

```
34: procedure poison-subtree(activation record a, channel c)
35:   mark a as poisoned
36:   if a has children then
37:     for each child s that knows about c do
38:       poison-subtree(s, c)
39:   else                                at a leaf: propagate its death
40:     for each channel c known to a do
41:       evaluate-channel(a, c)
42:     collect-poisoned(a)
43: procedure collect-poisoned(activation record a)
44:   while a is an activation record do
45:     for each child c of a do
46:       return if c is not poisoned or has children
47:     if a has children then
48:       terminate a's children
49:       if a is not poisoned then
50:         set a's state to its handler
51:         schedule a                                 run the catch body
52:         return
53:     for each child c of a do
54:       evaluate-channel(a, c)
55:     a ← a's parent
56: function channel-poisoned(activation record a, channel c)
57:   return false if a is not poisoned
58:   for each child s that knows about c do
59:     return true if channel-poisoned(s, c)
60:   return true if no child knows about c, false otherwise
61: function channel-ready(activation record a, channel c)
62:   return false if a is poisoned
63:   for each child s that knows about c do
64:     return false if not channel-ready(s, c)
65:   return false if no child knows about c, true otherwise
66: procedure transmit(value v, activation record a, channel c)
67:   if a has no children then
68:     set the value of c in a to v
69:     mark a as unblocked
70:     schedule a
71:   else
72:     for each child s of a that knows about c do
73:       transmit(v, s, c)
```

Figure 10: The runtime algorithms (2/2)

transmit is called (lines 66–73, called in line 31) that walks down the tree to perform the communication (line 68).

Otherwise, if no child is poisoned and we are not at the top of the tree, *evaluate-channel* moves up the tree (lines 32–33).

The *collect-poisoned* procedure (lines 43–55) walks up the tree, looking for a node with all of its children poisoned. The children of such nodes are terminated and the process proceeds up the tree unless the thread itself is not poisoned (line 49). When all a node's children are poisoned but it is not, it indicates that the node caught the exception and can handle it. Thus, the state of the thread is set to that of the exception's handler (line 50) and the thread is scheduled (line 51).

Otherwise, the effect of the children terminating is propagated (lines 53–54) and *collect-poisoned* continues to walk up the tree.

7 Related Work

We first discuss the relationship of SHIM to existing work on data races in concurrent systems, then compare SHIM to existing concurrent programming languages.

7.1 Data Races

There is a growing literature on data races in concurrent programming languages, including work on type systems and static analysis tools to detect races [17, 19, 9], dynamic checkers [32, 13, 18], and language constructs and restrictions [3, 34].

A race condition occurs when two threads simultaneously access the same data variable and at least one of the accesses is a write. SHIM simply prohibits such races. First, concurrent accesses to the same data variable must be guarded by *next* instructions that forces the accesses to be synchronized, thus deciding the sequence of read and write accesses. Second, at most one thread owns each shared variable at a time:

no thread but the owner thread may write the value of the variable.

In SHIM, to enable concurrent writes, the owner thread must implement a deterministic arbiter that gathers tentative write orders from concurrent threads and deterministically decides what to do. Designing arbiters typically requires some careful, domain-specific thinking, but it can be done. We presented one such arbiter in Section 4.

In fact, such an arbiter is exactly what is required from, say, a Java programmer to make his program behave properly. Hence, the distinction between SHIM and Java is that in the absence of a deterministic arbiter, the SHIM compilers reject the program, whereas the Java compiler produces a nondeterministic program.

Many authors argue that the absence of data races does not imply the absence of concurrency-related bugs [12, 19, 2, 34], and we agree. All these projects share a common view: the execution of a concurrent program may produce undesirable behaviors arising from the interleaving of execution steps in concurrent threads (not necessarily reads and writes to the same data variable). They draw clever lines between acceptable and unacceptable interleavings and tackle the second kind. Importantly, SHIM not only enforces mutual exclusion in concurrent accesses to shared data variables but in general prohibits all interleaving-dependent behaviors with the obvious drawback it is more restrictive and the obvious benefit of simplicity.

This does not solve all problems however. For instance, in an example from Vaziri et al. [34], updates to the *zipcode* and *city* fields of a *customer* object should be constrained so that concurrent updates may not end up with an inconsistent state: the zipcode from update (a), with the city from update (b). In SHIM, such an atomicity property can be enforced by making sure the same thread is responsible for updating these two fields, hence a common arbiter is used. We have no doubt such higher-level concerns are very relevant to the "correct" behavior of concurrent programs. Determinism as such is only a tool that can contribute to correctness. It cannot decide high-level atomicity constraints assumed by the programmer, but does make their implementation easier.

7.2 Concurrent Programming Languages

SHIM is hardly the first concurrent language to be proposed [1], but most others use more error-prone communication mechanisms. For example, the shared-memory-and-monitors style used in Java and C# first appeared in the mid-1970s in Brinch Hansen's concurrent Pascal [10]. Evolving as they did from the desire for a high-level language for programming operating systems, monitors were just designed to provide a universal synchronization mechanism, not the least error-prone mechanism. Even Brinch Hansen states "first-in, first-out queues are indeed more convenient to use [than monitors]" [11, p. 39].

Hoare's CSP was one of our inspirations [23], in particular its choice of rendezvous-style communication, which has been adopted by such languages as OCCAM [25] and Ada [24]. Both, however, provide nondeterministic selection among multiple events, which can create a race.

Our style of determinism was inspired by Kahn's little language [27], which prohibits nondeterministic merges and therefore provides race-free concurrency. Our original SHIM language [15] was very closely based on Kahn's ideas, albeit with rendezvous-style communication to avoid the challenges of scheduling Kahn networks in bounded memory [30].

Aspects of SHIM, in particular its philosophy of determinism, were inspired by the now-large body of work on synchronous programming languages [4, 5], especially Esterel [7] which is imperative. However, the execution model in SHIM is deliberately asynchronous and designed to handle widely varying execution rates; only communication is synchronous.

Exceptions in SHIM, while closely matching the priority rules of concurrent exceptions in Esterel [6], also largely differ from those due to the absence of a master clock. In SHIM, exceptions only propagate with communications if and when they take place, rather than unconditionally at instant boundaries. In particular, in Section 3.4, note that an exception in SHIM typically kills a FIFO only after it has emptied, whereas similarly structured code in Esterel would kill the FIFO "now" discarding all values in transit.

Fair threads [8] also based on synchronous programming ideas address concerns similar to ours but require the a priori choice of a scheduling policy, rather than providing behaviors independent from it.

To a lesser degree, SHIM was also inspired by the join-calculus [20], which tries to force data-locality amenable to efficient implementation into Milner's π-calculus [28].

8 Conclusions and Future Work

We have presented SHIM, a practical little language that provides Kahn-like deterministic concurrency in a traditional imperative-language setting. In addition to arithmetic expressions and classical control-flow constructs, we provide recursive procedure calls, synchronized shared variables, and exceptions, all in a concurrent setting. We proposed a core syntax defined the semantics of this little language, and described an unoptimized implementation of our language as a translation to C. Our compiler, which is roughly 2000 lines of OCAML, produces functioning code for every example we presented in this paper except those with non-scalar types.

The syntax we propose, while rich enough to express interesting programs, is just a skeleton on which we are building a complete language. In particular, we shall address the lack of data structures in SHIM. Because pointers may introduce inter-thread aliasing and the potential for races, this is not straightforward. We think a mix of user specified annotations and inference is required here, using aliasing analysis and ownership types [14] as basic building blocks.

Making our language practical will also require a proper module or package system for encapsulating libraries. By nature, however, the design of this aspect of the language is largely orthogonal to the semantics we have presented here and we expect a system from another successful language can be employed with few problems.

From a more theoretical viewpoint, we would like to formalize and prove that our language is deterministic, i.e., prove a confluence property for our semantics. Our adherence the Kahn principle in the design of our language strongly suggests this should be true and possible to prove.

Boundedness was a goal of the original SHIM language [15] that the language in this paper does not guarantee since it permits unbounded recursion. While this is very convenient for certain software systems, it makes a direct hardware implementation difficult. In order to ensure decidability of type-checking proof assistants such as Coq [33] require that functions are provably terminating, total, and deterministic. We plan a mechanism for hardware implementation of SHIM that will involve similar constraints and techniques.

In short, we believe we have a solid, powerful foundation for expressing concurrent algorithms for both software and hardware. However, much remains to be done.

References

[1] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[2] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *Proceedings of the Workshop on Verification and Validation of Enterprise Information Systems (VVEIS)*, pages 82–93, Angers, France, April 2003.

[3] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 382–400, Minneapolis, Minnesota, October 2000.

[4] Albert Benveniste and Gérard Berry. The synchronous approach to reactive real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[6] Gérard Berry. Preemption in concurrent systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93, Bombay, India, December 1993. Springer-Verlag.

[7] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[8] Frédéric Boussinot. FairThreads: mixing cooperative and preemptive threads in C. RR 5039, INRIA, 2003.

[9] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 56–69, Tampa Bay, Florida, October 2001.

[10] Per Brinch Hansen. The programming language Concurrent-Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[11] Per Brinch Hansen. Monitors and concurrent Pascal: A personal history. In *History of Programming Languages II*, pages 1–35, Cambridge, Massachusetts, April 1993.

[12] Michael Burrows, K. Rustan, and M. Leino. Finding stale-value errors in concurrent programs. Technical Report 2002-004, Systems Research Center, Compaq, May 2002.

[13] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, pages 258–269, Berlin, Germany, June 2002.

[14] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64, Vancouver, British Columbia, Canada, October 1998.

[15] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

[16] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, 14(8):854–867, August 2006.

[17] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, pages 219–232, Vancouver, British Columbia, Canada, June 2000.

[18] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 256–267, Venice, Italy, January 2004.

[19] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, pages 338–349, San Diego, California, June 2003.

[20] Cedric Fournet and Georges Gonthier. The join calculus: a language for distributed mobile programming. In *Applied Semantics. International Summer School (APPSEM)*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332, Caminha, Portugal, August 2002. Springer-Verlag.

[21] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, New Jersey, 1986.

[22] Rajiv Gupta and Mary Lou Soffa. The efficiency of storage management schemes for Ada programs. In *Proceedings of the ACM SIGAda International Conference on Ada*, pages 164–172, Paris, France, May 1985.

[23] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

[24] Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heliard. Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(6b):1–261, June 1979.

[25] INMOS Limited. *occam 2 Reference Manual*. Prentice Hall, 1988.

[26] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C,. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, California, June 2002.

[27] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[28] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[29] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Portland, Oregon, January 2002.

[30] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995. Available as UCB/ERL M95/105.

[31] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Åarhus, Denmark, 1981.

[32] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[33] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA. http://coq.inria.fr/doc/main.html.

[34] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 334–345, Charleston, South Carolina, January 2006.