Scheduling-Independent Threads and Exceptions in SHIM

Olivier Tardieu
Department of Computer Science
Columbia University, New York
tardieu@cs.columbia.edu

Stephen A. Edwards*
Department of Computer Science
Columbia University, New York
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, typical 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 deterministic concurrent language, 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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms

Languages, Theory

Keywords

Hardware/software codesign, Deterministic model of computation

1. INTRODUCTION

Embedded systems differ from traditional computer systems in their need for concurrent descriptions to handle simultaneous activity in their environment or to exploit parallel, often heterogeneous hardware. While programming such systems in traditional sequential languages would be convenient, it greatly hinders exploiting parallelism. Instead, we propose a concurrent language whose constructs simply avoid many 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 [24] and the CCured rewriter [27] 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 propose 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 do make it impossible to implement certain algorithms that appear nondeterministic but are actually well-behaved, we believe SHIM is both expressive and amenable to efficient implementation. We argue for SHIM's expressivity in a series of examples and describe a preliminary compiler able to generate single-threaded C code.

1.1 The New SHIM

Our first 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 for the same reasons as Kahn's networks [25], which they resemble by design, but are simpler to schedule and require only bounded resources by adopting rendezvous-style communication inspired by Hoare's CSP [21].

*Edwards and his group are supported by the NSF, Intel, Altera, the SRC, and NYSTAR.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

e ::= $L \mid V \mid op_1\ e \mid e\ op_2\ e \mid$ `(` e `)` **expressions**

s ::= V `=` e`;` $\mid P$`(` $(V(,V)^*)?$ `);` **statements**
$\mid$ `{` b^* `}` $\mid$ `if (` e `)` s `else` $s \mid$ `while (` e `)` s
$\mid s$ `par` $s \mid$ `next` V`;` $\mid$ `try` s `catch(` E `)` $s \mid$ `throw` E`;`

b ::= $T\ V$`;` $\mid s$ **block statements**

d ::= $T\ V \mid T$ `&`V **parameter declarations**

p ::= `void` P`(` $(d(,d)^*)?$ `) {` b^* `}` **procedure declarations**

m ::= p^* `void main() {` b^* `}` **programs**

L denotes literals, T types (e.g., `int`, `void`), E exceptions, V variables, and P procedures. `par` binds most tightly.

Figure 1: The syntax of SHIM

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 proposed [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 language. The result resembles C and Java (and can be used as such) while still guaranteeing deterministic concurrency without requiring careful attention to the use of, e.g., the semaphores found in many concurrent programming languages. Instead of a static network of processes connected by predefined point-to-point communication channels, SHIM now allows the dynamic creation of threads and "shared variables." Using recursion, one can instantiate arbitrarily many threads. We also introduce concurrent, deterministic exceptions that resemble those in Java (and have the same semantics in single-threaded code), but also behave deterministically with concurrency, providing a powerful, structured way to control the execution of concurrent threads, a major omission in many 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*.

```
void 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 one *void* parameter to the procedure (used as a synchronization barrier) and two lines of code turns this depth-first search into a concurrent breadth-first search:

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

In Section 4, we augment this code to return the 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 functionality guarantees about complete designs.

We first introduce SHIM (Section 2) and exceptions (Section 3). We develop the breadth-first-search example in Section 4, provide the formal semantics of SHIM in Section 5, and describe a basic compiler in Section 6. We discuss related work in Section 7.

2. THE BASIC SHIM LANGUAGE

SHIM, whose syntax is summarized in Figure 1, draws from 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.

SHIM adds four constructs to the usual imperative statements:

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 responsible for running *p* and *q* and suspends the current thread until the completion of both. As a result, a parent never runs when any of its children are running.

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 q* may behave differently, just as {*p*;} *q* and and *p*; *q* may be different in C.

The SHIM scheduler may interleave concurrent threads in any way that does not violate inter-thread communication rules. Concurrent threads thus run asynchronously.

To achieve behavior independent of arbitrary scheduling decisions, we impose restrictions on "shared variables." In a *par*, each variable may be implicitly passed to at most one thread by reference, but many threads may have the same variable passed by value.

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

```
void f(int &x) {}     // pass-by-reference
void g(int x) {}      // pass-by-value
void main() {
  int a; a = 0; int b; b = 0;
  a = 1; par b = a;   // OK
  a = 1; par a = 2;   // Incorrect: 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);     // Incorrect: a is an lval twice
}
```

An *lval* is passed by reference; an *rval* by value. E.g.,

```
void main() {
  int a; a = 3; int b; b = 5; int c; c = 1;
  {               // a is passed by reference
    a = a + c;    // a = 4, b = 5, c = 1
    a = a + b;    // a = 9, b = 5, c = 1
  } par {         // b is passed by reference
    b = b - c;    // a = 3, b = 4, c = 1
    b = b + a;    // a = 3, b = 7, c = 1
  }               // a = 9, b = 7, c = 1
}
```

2.2 Communication

That no variable may be passed by reference to more than one thread simultaneously prevents a thread from accidentally modifying another thread's copy of a variable. Instead, the *next* instruction forces threads to synchronize before performing inter-thread communication. *Next* transmits a variable's value when it is a pass-by-reference parameter and receives it otherwise. For example,

```
void f(int a) {  // a is a copy of c
  a = 3;
  next a;        // synchronize with g; a gets c's value
                 // a = 5
}
void g(int &b) { // b is an alias for c
  b = 5;
  next b;        // synchronize with f
                 // b = 5
}
void 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* behaves as a send operation in *g* and *next a* behaves as receive operation in f. Together, these transmit *c*'s value to *a*.

To make communication deterministic, a *next* instruction forces all threads sharing the variable to synchronize. E.g., in the previous example, *next a* and *next b* execute simultaneously.

Such synchronization may cause deadlocks. For example,

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

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

In SHIM, *void* variables provide pure synchronization.

A thread is only required to synchronize on a variable it shares. Moreover, if a thread terminates, it is no longer compelled to participate in a synchronization and therefore does not cause a deadlock. For example,

```
void main() {
  void a; void b;
  { next a; next b; } par { next b; } // no deadlock
  { next a; next a; } par { next a; } // no deadlock
}
```

does not deadlock.

Pending synchronizations may take place in any order. In

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

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

2.3 Delegation

If a thread spawn 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,

```
void 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,

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

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

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

In general, 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 passed around a 1, then a 2, a 3, etc.

```
void 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
  }
}
void g(int b, int &c) {
  while (true) {
    next b;      // receives
    c = b;
    next c;      // sends
  }
}
void main() {
  int a; a = 0; int b; int c;
  f(a, b); par g(b, c); par g(c, a);
}
```

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

```
void 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"), but doing so can be awkward. SHIM's exception mechanism is layered on top of 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,

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

Exceptions may be raised from inner threads. Whether concurrently running threads in the scope of the exception are affected depends on communication. For example,

```
void 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, so thread 1 has no way to interrupt thread 2 at a deterministic point in its execution (e.g., at a particular value of *i*). Thread 2 runs forever. The compiler warns about such patterns.

A thread is *poisoned* iff it raises or propagates an exception. If a thread attempts to communicate with a poisoned thread, it also becomes poisoned, thus propagating the exception. For example,

```
void 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 = 5, j = 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. An attempt to communicate with a dying thread poisons the thread attempting the communication. For example,

```
void main() {
  void a; void b; void c;
  try {               // thread 1:  shares a, b, and c
    {                 // thread 1a: shares a
      a;
    } par {           // thread 1b: shares b
      b;
      throw T;
    } par {           // thread 1c: shares c
      while (true) {  // runs forever
        next c;       // synchronize with thread 4
      }
    }
  } par {             // thread 2:   shares a
    next a;           // is poisoned by thread 1
  } par {             // thread 3:   shares b
    next b;           // is poisoned by thread 1b
  } par {             // thread 4:   shares c
    while (true) {    // runs forever
      next c;         // synchronize with thread 1c
    }
  } catch(T) {}
}
```

Here, thread 3 gets poisoned while attempting to communication with thread 1b. Thread 4 communicates with thread 1c, which runs normally. Thread 2 attempts to communicate first with thread 1a then with thread 1 itself since, upon the completion of thread 1a, thread 1 resumes responsibility for *a*. Thread 1 is dying due to exception *T* in thread 1b. Therefore, thread 2 gets poisoned. Here again, we observe that thread 1, while "dying," never actually dies since thread 1c never returns.

3.1 Scopes and Priorities

The rules for our exceptions deliberately follow those in Esterel [7, 6]. Poison does not flow outside the scope of the exception:

```
void main() {
  int i; i = 0; int j; j = 0;
  {                 // thread 1
    try {           // thread 1a
      i;
      throw T;
    } par {         // thread 1b
      i = i + 1;    // is executed
      next i;       // is poisoned by thread 1a
      i = i + 1;    // is not executed
    } catch(T) {}
  } par {           // thread 2
    j = j + 1;      // is executed
    next i;         // executes normally
    j = j + 1;      // is executed
  }                 // i is 1, j is 2
}
```

Here, 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 synchronize with the *next* instruction in thread 1b if *T* had not been raised, instead blocks until the completion of the *try-catch* block, at which point it may take place as usual, since thread 2 is then the only remaining thread sharing *i*.

Second, when exceptions are raised in parallel, the outermost exception takes priority and defines the exit point. For example,

```
void main() {
  int i; i = 1;
  try {
    try {
      throw T;
    } par {
      throw U;
    } catch(T) { i = i * 2; } // handler is not executed
    i = i * 3;                // is not executed
  } catch(U) { i = i * 5; }   // i = 5
}
```

3.2 The FIFO Revisited

An exception allows us to elegantly terminate our FIFO example.

```
void source(int &a) {
  while (a > 0) {
    a = a - 1;
    next a;     // sends a
  }
  throw T;
}
void sink(int b) {
  while (b != 0) {
    next b;     // receives b
  }
  // do something else
}
void 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* procedure sends 4, 3, 2, 1, and 0 to the three-place FIFO, which 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. Because the sink is not part of the scope of T, it is unaffected by T.

Exceptions make it unnecessary to add an explicit termination condition or end-of-stream data token in our FIFO.

4. BREADTH-FIRST SEARCH EXAMPLE

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, SHIM could easily accommodate a type for binary trees:

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

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

In general, pointers can break the determinism of SHIM. However, in the examples below we shall only read from binary trees, avoiding any races. We plan a more complete discussion of data structures and pointers in SHIM in future work.

4.1 The Membership Algorithm

First, consider a sequential algorithm for deciding membership of a key in a tree.

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

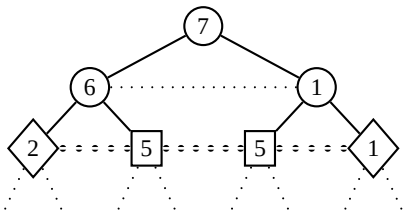
This explores the leftmost branch first, reports success through the *Found* exception, and terminates normally on a finite tree that does not contain the key.

To turn this procedure into a concurrent breadth-first search in SHIM, we add a *void* variable for synchronization, a *next* instruction, and a *par*.

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

The *par* runs the two recursive calls concurrently. The *next* statement allows a branch that finds the *key* to interrupt concurrently-running branches by forcing them to synchronize.

Consider finding the 5 key in the tree below.



The horizontal dashed lines indicate where recursive calls at the same depth synchronize. As a result, all breadth-first search procedures terminate at depth 3 either by raising exception *Found* (the square nodes) or by getting poisoned (the diamonds). Nodes at depth 4 and below are not visited.

Incidentally, because all threads share the key without modification, we could instead synchronize on it, allowing us to write:

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

4.2 The Lookup Algorithm

We now augment the search algorithm to return the value associated with the key. The obvious thing is to add a pass-by-reference parameter for returning the value:

```
void 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 because *value* is passed by reference to both parallel recursive calls, which may cause a race. Consider again the example in the previous section. It contains the key in two nodes at depth 3. Which value should be returned? The leftmost or the rightmost value? The above piece of code does not specify a deterministic behavior and consequently our compiler rejects it.

However, carefully combining concurrency, recursion, and exceptions allow us to implement a deterministic concurrent breadth-first-search algorithm:

```
void 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; }
  }
}
```

This introduces a 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, nested exceptions implement priorities. There are four cases to consider:

1. Neither branch raises exception *Found*: the procedure terminates normally or runs forever if the tree is infinite.
2. Only the left branch raises *Found*: the exception kills the right branch and propagates upwards. The variable *value* contains the value returned by the left branch.

3. Only the right branch raises *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 *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.

Therefore, the *assoc* procedure always 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.

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 possible transitions between program states. For simplicity, we first present SHIM without exceptions and add them later in Section 5.4.

Choosing an appropriate notion of state is fundamental. We begin with an informal description of the components of a state, then describe our state encoding, explain our semantic rules, and finally show how they execute an example.

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 shared variable location, i.e., the source location of the data copied in a *next* operation.

Although our semantics refers to a shared global store for simplicity, access to the store is disciplined enough to allow it to 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 Notation

We assume the parameters, local variables of a thread, and procedure names are distinct. 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. $\mathrm{body}(p)$ denotes the sequence of statements in p.

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

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 3 has root $\mathbf{0}|\{\mathtt{c} \mapsto \lambda\}$ and leaves `f(c);`$|\{\mathtt{c} \mapsto \mu, \lambda\}$ and `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 $\mathrm{Def}(r) = \mathrm{Def}(v)$.

5.2 Formal Semantics

In Figure 2, we formalize the semantics of SHIM without exceptions as a set of deduction rules in a structural operational style [29].

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 now; we shall discuss it in Section 5.4.

The rules for *if* and *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 order. It resolves local variable and parameter names to values using the function $\sigma \circ v_{\mathrm{loc}}$.

At a local variable declaration, we bind the name of the variable to a new location whose value starts at $\bot$. 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 procedure calls, blocks, and parallel branches. Rule *join* handles one branch at a time so that a branch releases its shared variables 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 $v_{\mathrm{glb}}(x)$ and expects $v_{\mathrm{loc}}(x)$ to be updated to reflect the current value of $v_{\mathrm{glb}}(x)$. 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.

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

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

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

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

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

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

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

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

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

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

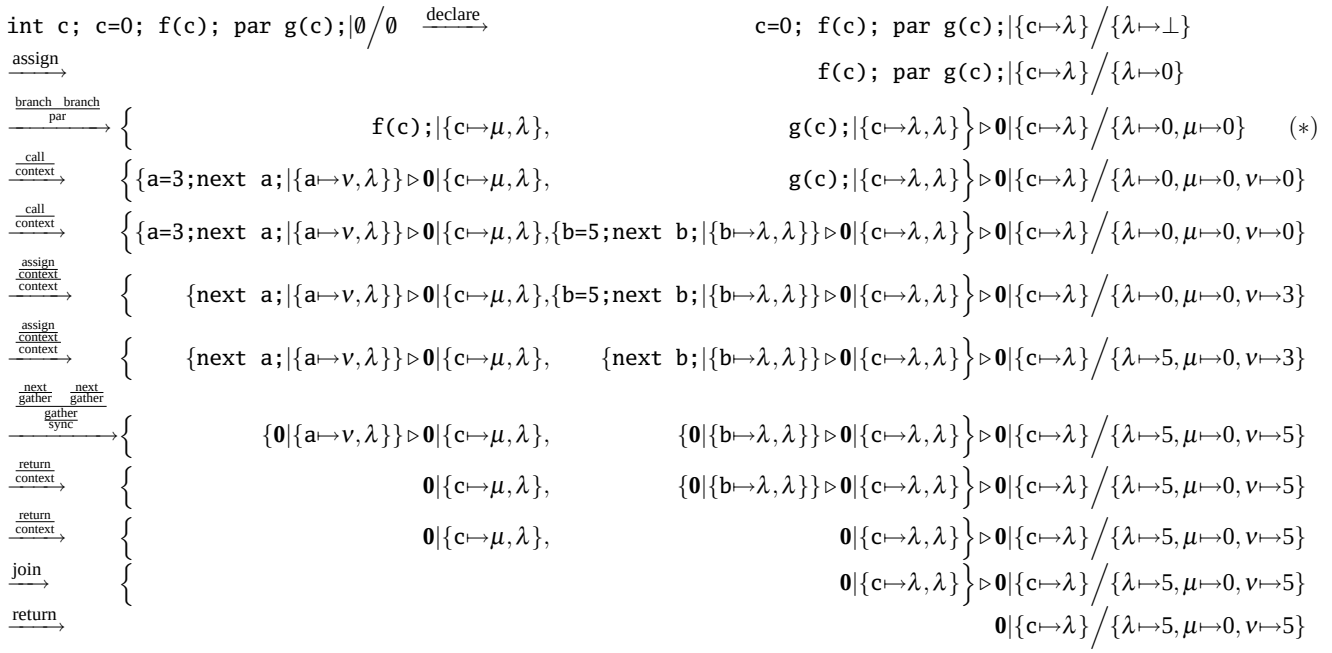


Figure 3: An example of execution.

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 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.

5.3 Example

Figure 3 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 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 concurrent assignments may occur in any order. After the assignments, the threads synchronize and the value at location v receives the value at location λ. Finally, both procedures and both branches return and the program terminates.

5.4 Exceptions

In Figure 4, 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 track exceptions scopes in the semantics, we augment the residual structure. We add an extra piece of information *m* atop $\triangleright$ symbols, 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.

Rule *try* 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 a copy of 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 poisoned. Second, rule *throw2* marks enclosing threads with the same **X** to indicate they are dying. Finally, the *throw3* rule stops the poison at the boundary of the exception scope. Rule *exit* handles the completion of poisoned threads. When all subthreads of a thread are poisoned, they are deleted. Rules *handler* and *skip-handler* take care of exceptions handlers. Rules *exception*, *exception2*, and *exception3* decide when a location λ is poisoned by a thread *r*: $\lambda \in \mathrm{Exc}(r)$. Rules *next-fail*, *gather-fail*, *sync-fail* propagate poison from poisoned or dying threads to threads that attempt to communicate with them.

5.5 Determinism

We claim 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: each thread of control can block on at most one communication at once and cannot retreat from an attempt to communicate.

Informally, whenever several transitions are possible from one program state, they commute: they may be applied in any order and all permutations will result in the same final state. However, due to the size of the semantics, we do not have a formal proof.

6. A BASIC IMPLEMENTATION

Our compiler, which is roughly 2000 lines of OCAML, generates single-threaded C code from SHIM programs. It produces functioning code for every example we presented in this paper except those with non-scalar types.

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 *next* statements. 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.

Although threads in SHIM are properly nested, procedure activation records cannot always be stored on a stack. Instead, our runtime system maintains a tree of activation records on the heap (using *malloc* and *free*). Each record has pointers to its child, next sibling, and parent (caller) records that allow the tree to be traversed when procedures communicate, terminate, or throw exceptions.

Each activation record includes a pointer to the C function that implements its SHIM procedure, a field that holds the control state (equivalent to a program counter) between invocations of the function, an indication of the exception that was thrown, if any, and the control state for the function if an exception is caught.

Finally, each activation record contains an array of information about the channels the procedure is connected to (knows about). Each entry has a pointer to the value of the corresponding variable, a flag that indicates whether the procedure is blocked on the channel, and a reference to its parent channel.

The central scheduler is straightforward: it simply takes a pointer to an activation record off a stack of runnable threads and calls the function whose pointer is in the activation record.

The behavior of the *next* and *throw* statements are more complex. *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. When a thread terminates, it removes itself from the activation record tree and unblocks any threads that were waiting for it to communicate.

7. RELATED WORK

We discuss how SHIM relates to work on data races in concurrent systems, then compare it to other concurrent 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 [30, 13, 18], and language constructs and restrictions [3, 32].

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 force 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: only the owning thread may set the value of the variable.

Concurrent writes in SHIM require the owning thread to implement a deterministic arbiter that gathers tentative write orders from other 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 predictably. Hence, the

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4: The semantics of exceptions in SHIM.

distinction between SHIM and Java is that in the absence of a deterministic arbiter, the SHIM compilers reject the program, whereas a 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, 32], and we agree. All these projects hold that 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 lines between acceptable and unacceptable interleavings and address the latter. SHIM not only enforces mutual exclusion in concurrent accesses to shared data, it prohibits all interleaving-dependent behavior: a simpler but more restrictive approach.

This does not solve all problems, however. For instance, in an example due to Vaziri et al. [32], 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 the first update with the city from the second. In SHIM, such atomicity can be enforced by ensuring a single thread is responsible for updating both fields, necessitating an arbiter. 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 automatically enforce all 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 designed as a universal synchronization mechanism, not the least error-prone. Even Brinch Hansen states "first-in, first-out queues are indeed more convenient to use [than monitors]" [11, p. 39].

Hoare's CSP [21] inspired us, in particular its rendezvous communication, which has been adopted by such languages as OCCAM [23] and Ada [22]. Both, however, provide nondeterministic selection among multiple events, which can create a race.

Our style of determinism was inspired by Kahn's little language, which prohibits nondeterministic merges and therefore provides race-free concurrency [25]. 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 [28].

Aspects of SHIM were inspired by the now-large body of work on synchronous languages [4, 5], especially the imperative Esterel language [7]. But the execution model in SHIM is 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], are very different because of the absence of a master clock. In SHIM, exceptions only propagate with communications if and when they take place, rather than unconditionally at clock cycle boundaries. In particular, in Section 3.2, 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] address concerns similar to ours but require an a priori choice of 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 [26].

8. CONCLUSIONS AND FUTURE WORK

We have presented SHIM, a practical imperative language that provides Kahn-like deterministic concurrency. 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 language, and described an unoptimized implementation of our language as a translation to C.

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, adding them is not straightforward. We plan to use a mix of user-specified annotations and inference; aliasing analysis and ownership types [14] will be the 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. 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. 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 hardware implementation difficult. To ensure the decidability of type-checking, proof assistants such as Coq [31] require functions be 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.

9. REFERENCES

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

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

[3] D. F. Bacon, R. E. Strom, and A. 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, Oct. 2000.

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

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan. 2003.

[6] G. 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, Dec. 1993. Springer-Verlag.

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

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

[9] C. Boyapati and M. 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, Oct. 2001.

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

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

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

[13] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, pages 258–269, Berlin, Germany, June 2002.

[14] D. G. Clarke, J. M. Potter, and J. 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, Oct. 1998.

[15] S. A. Edwards and O. 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, Sept. 2005.

[16] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, 2006. To appear.

[17] C. Flanagan and S. 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] C. Flanagan and S. 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, Jan. 2004.

[19] C. Flanagan and S. 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] C. Fournet and G. 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, Aug. 2002. Springer-Verlag.

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

[22] J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, J. G. P. Barnes, O. Roubine, and J.-C. Heliard. Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(6b):1–261, June 1979.

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

[24] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C,. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, California, June 2002.

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

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

[27] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Portland, Oregon, Jan. 2002.

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

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

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

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

[32] M. Vaziri, F. Tip, and J. 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, Jan. 2006.