Scheduling-Independence in SHIM

Olivier Tardieu

Columbia University, New York

joint work with Prof. Stephen A. Edwards

Software/Hardware Integration Medium

SHIM is a concurrent programming language

- C/Java-like syntax and semantics for sequential code
- no aliasing
- new constructs for concurrency

$\Rightarrow$ portable thread semantics

for embedded systems and multicore CPUs

- including control systems (automotive, avionics...), DVD players...
- excluding web servers, sensor networks, scientific computing...

Related Work

C, JAVA, and “safe” Java dialects

- Sequential constructs and semantics

Synchronous programming languages

- SHIM is asynchronous à la Kahn networks
- SHIM has synchronous communications à la CSP
- SHIM has streams, simple causality à la Lustre
- SHIM is imperative, has exceptions à la Esterel
- SHIM is dynamic à la Boussinot

Portability: from C to Java

C exposes the architecture to the programmer

- range of integers
- evaluation order: `subtract(i++, i++);`
- out-of-bound array access...

Java is meant to be portable

- 32-bit integers
- fixed evaluation order
- array bound checking...

Portability: from Java to SHIM

Multithreaded Java programs are not portable

- platform-dependent scheduling (compiler, runtime, OS, hardware)
- arbitrary accesses to shared variables $\Rightarrow$ data races
- arbitrary locking schemes $\Rightarrow$ deadlocks

SHIM guarantees portability

- no data races
- deterministic reproducible deadlocks
- output data streams independent from scheduling policy (Kahn)

Outline

- Syntax
- Breadth-First Search
  - Concurrency
  - Synchronization
  - Exceptions
  - Shared variables?
- FIFO
  - Communication
- Formal Semantics
- Conclusions

Syntax

Core C with procedures rather than functions

- pass-by-value and pass-by-reference '&' parameters
- no return value

Java-like sequential semantics: evaluation order...

New constructs

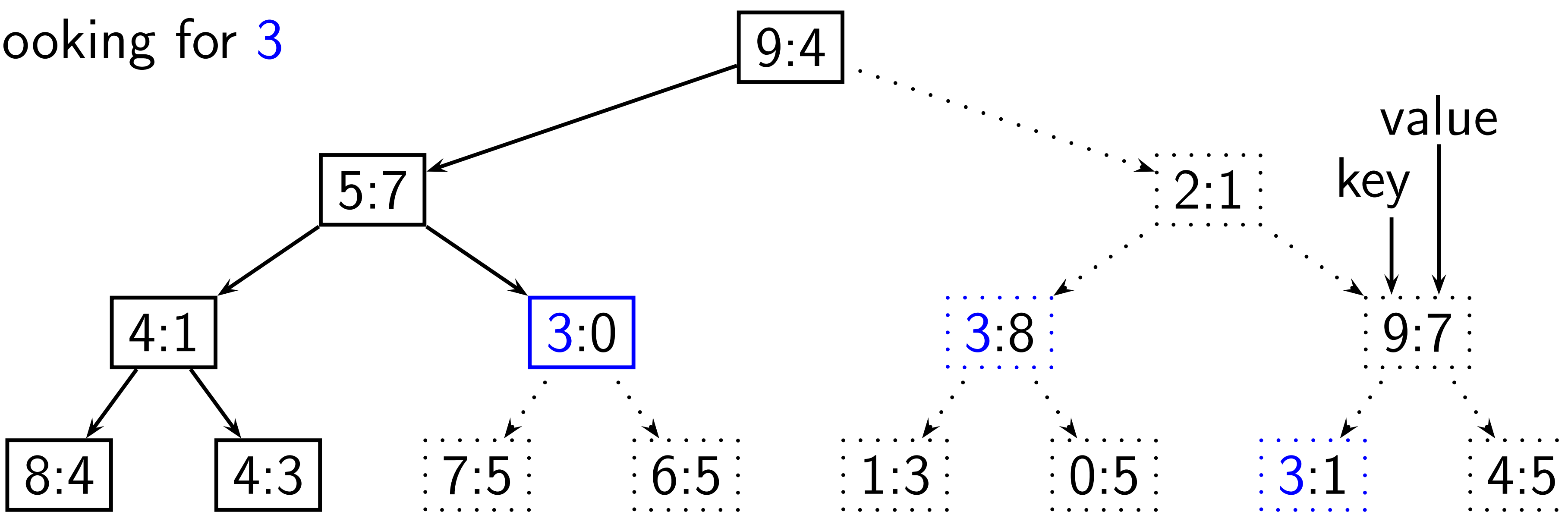
- *stt* **par** *stt*   for concurrency
- **next** *var* ;   for synchronization and communication
- **try** *stt* **catch**(*exc*) *stt*   to define and handle exceptions
- **throw** *exc* ;   to raise exceptions

Depth-First Search

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

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

Looking for 3



Concurrent Search?

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

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

```
void breadth_first_search(int key, Tree tree) {
  if (tree == null) return;
  if (key == tree.key) throw Found(tree.value);
    breadth_first_search(key, tree.left);
  par                                          // fork threads
    breadth_first_search(key, tree.right);
}
```

Parallel branches execute asynchronously (arbitrary scheduling)

Problems: multiple key occurrences? termination?

Synchronization and Exceptions

```
void assoc(int key, Tree tree, void tick) {
  if (tree == null) return;
  if (key == tree.key) throw Found(tree.value);
  next tick;                                          // sync threads
    assoc(key, tree.left, tick);
  par                                                 // fork threads
    assoc(key, tree.right, tick);
}
void breadth_first_search(int key, Tree tree) {
  void tick; assoc(key, tree, tick);
}
```

The next instruction forces threads to synchronize

Exceptions propagate at synchronization points

$\Rightarrow$ The topmost occurrences of the key have priority

Problem: multiple key occurrences at the same level?

Synchronization

Mismatched synchronizations cause deadlocks

- `{ next a; next b; } par { next b; next a; }` `// deadlock`

  Thread 1 attempts to sync on a but thread 2 attempts to sync on b

Only live threads sharing the variable must synchronize

- `{ next a; next b; } par next b; par next a;` `// no deadlock`

  Thread 2 does not know about a

- `{ next a; next a; } par next a;` `// no deadlock`

  Upon completion of thread 2, only thread 1 knows about a

Synchronizations on distinct variables may occur in any order

- `next a; par next b; par next a; par next b; par next a;`

Deadlocks and Data Races

Data races

- are not easily detected
- lead to data corruption

Deadlocks

- are easily detected
- avoid data corruption

SHIM

- has no data races
- has reproducible deadlocks

Exceptions

```
void f() {
  void tick; int i = 0;
  try {                                          // thread 1
    next tick; throw T;
  } par {                                        // thread 2
    while(true) { i = i + 1; next tick; }
  } catch(T) { i = i * 3; }
}                                                // i = 6

void g() {
  void tick; int i = 0;
  try {                                          // thread 1
    next tick; throw T;
  } par {                                        // thread 2
    while(true) i = i + 1;                       // runs forever
  } catch(T) { i = i * 3; }
}                                                // never returns
```

Shared Variables?

```
void assoc(int key, Tree tree, void tick, int &value) {
  if (tree == null) return;
  if (key == tree.key) {
    value = tree.value;
    throw Found;
  }
  next tick;
    assoc(key, tree.left, tick, value);
  par                                          // possible race
    assoc(key, tree.right, tick, value);
}
```

Not legal in SHIM

- a variable can be passed by reference at most once in a par
- a variable can be passed by value without restriction in a par

Inference Rules

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

Lvals are passed by reference

Rvals are passed by value

A variable can be an lval at most once in a par

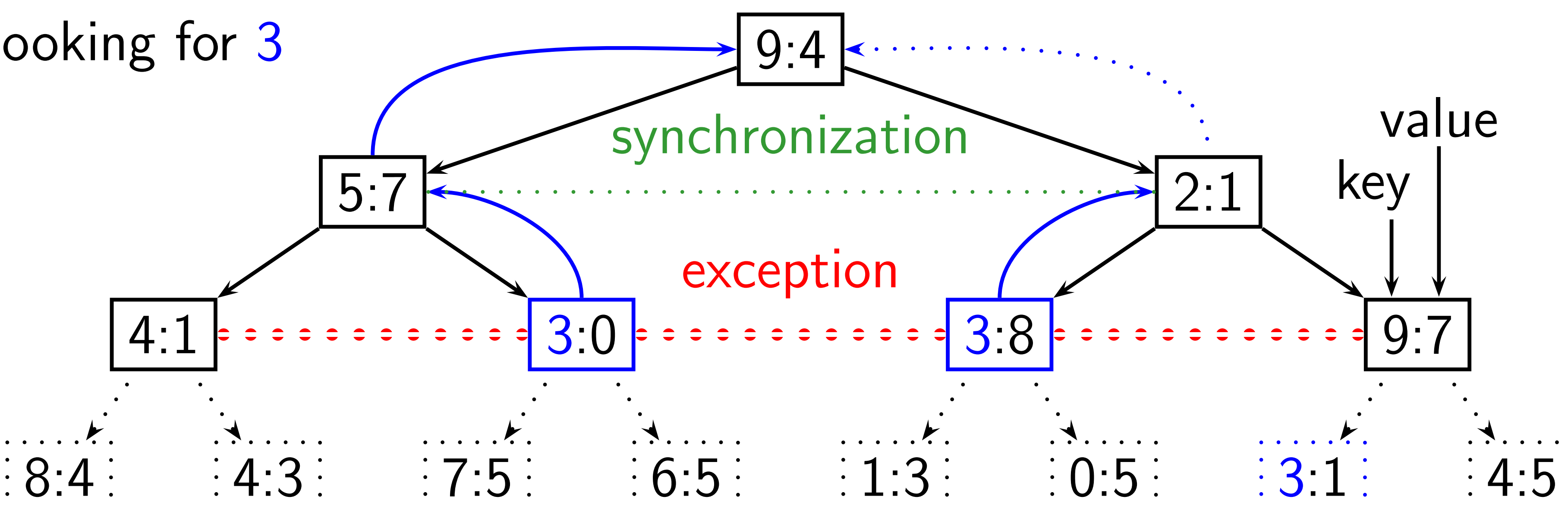
$\Rightarrow$ no shared memory $\Rightarrow$ no data race

Breadth-First Search Specification

Return the value of the topmost, leftmost key occurrence

- synchronize threads at each level
- kill concurrent threads if the exception is thrown
- return leftmost value if the exception is thrown multiple times

Looking for 3



Breadth-First Search

```
void assoc(int key, Tree tree, void tick, int &value) {
  if (tree == null) return;
  if (key == tree.key) {
    value = tree.value;
    throw Found;
  }
  next tick;
  int tmp = 0;
  try {
    assoc(key, tree.left, tick, value);
  } par {
    try {
      assoc(key, tree.right, tick, tmp);
    } catch(Found) { throw Right; }
  } catch(Right) { value = tmp; throw Found; }
}
```

$\Rightarrow$ The topmost, leftmost key occurrence has priority

FIFO

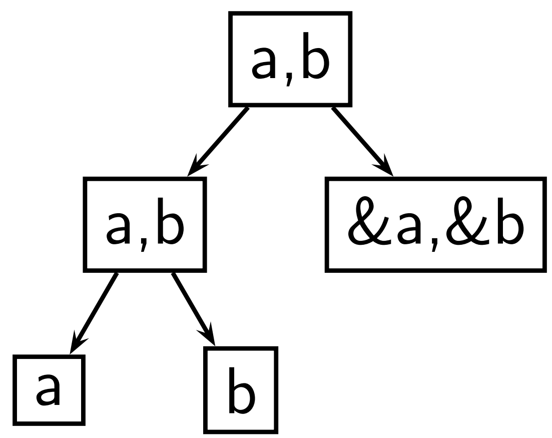
```
void fifo1(int input, int &output) {
  while(true) next output = next input;
}
void main() {
  int in = 0, out = 0;
  next in = 5; par fifo1(in, out); par next out;    // out = 5
}
```

Each variable declaration introduces a clocked stream of values

- next in lval position receives
- next in rval position sends
- communications take place at synchronization points
- variables passed by reference can be sent
- variables passed by value can only be received

Communication

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



Extending the FIFO

```
void fifo1(int input, int &output) {
  while(true) next output = next input;
}

void fifo(int n, int input, int &output) {
  if (n == 1) {
    fifo1(input, output);
  } else {
    int channel;
    fifo1(input, channel); par fifo(n - 1, channel, output);
  }
}
```

The data distribution in the fifo is unspecified

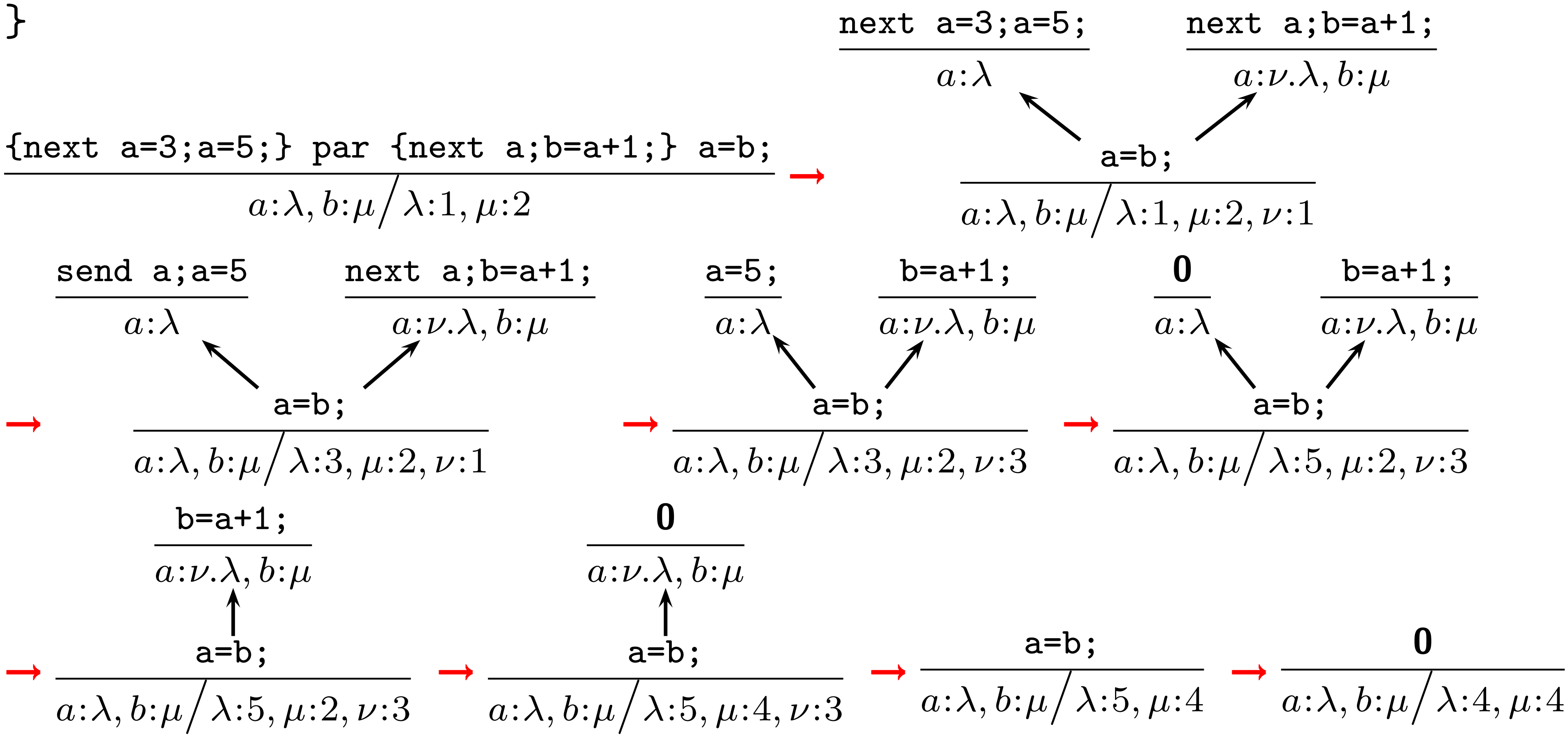
Disposing the FIFO

```
void source(int &a) {
  next a = 3; next a = 5; next a = 8; next a = 0;  // 0 is EOF
}
void sink(int b) {
  int i = 0;
  do i = i + next b; while (b != 0);
  ...
}
void main() {
  int a = 0, b = 0;
    try { source(a); throw T; } par fifo(3, a, b); catch(T) {}
  par
    sink(b);
}
```

The FIFO empties before terminating ($\neq$ Esterel)

Formal Semantics

```
void main() {
  int a = 1, b = 2;
  { next a = 3; a = 5; } par { next a; b = a + 1; }
  a = b;
}
```



Conclusions

SHIM guarantees scheduling independence

One cannot write unsafe programs in SHIM

One can write interesting safe programs in SHIM (local confluence)

Language-level scheduling-independent concurrency helps

- porting sequential programs to concurrent hardware
- optimizing implementations
- validating programs

A translator into sequential C code

Work in Progress

Mathematical proofs

- scheduling-independent behavior
- optimal exception propagation

Language extensions

- non-atomic arrays and graphs
- dynamic channel orientation

Implementation

- multicore code generation
- libraries
- experiments