Scheduling-Independence in SHIM

Olivier Tardieu

Columbia University, New York

joint work with Prof. Stephen A. Edwards
SHIM is a concurrent programming language

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

⇒ **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

- \texttt{stt \ par \ stt} for concurrency
- \texttt{next \ var;} for synchronization and communication
- \texttt{try \ stt \ catch(\textit{exc}) \ stt} to define and handle exceptions
- \texttt{throw \ exc;} to raise exceptions
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;
};
Concurrent Search?

```java
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
    breadth_first_search(key, tree.right);
}

// fork threads
```

Parallel branches execute asynchronously (arbitrary scheduling)
Problems: multiple key occurrences? termination?
void assoc(int key, Tree tree, \texttt{void} tick) { 
  if (tree == \texttt{null}) return;
  if (key == tree.key) throw Found(tree.value);
  next tick; // sync threads
  assoc(key, tree.left, tick);
  par
    assoc(key, tree.right, tick);
}
vvoid 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
⇒ The topmost occurrences of the key have priority
Problem: multiple key occurrences at the same level?
Synchronization

Mismatched synchronizations cause deadlocks

- \{ \text{next } a; \text{next } b; \} \text{ par } \{ \text{next } b; \text{next } a; \} \quad \text{// deadlock}

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

Only live threads sharing the variable must synchronize

- \{ \text{next } a; \text{next } b; \} \text{ par } \text{next } b; \text{ par } \text{next } a; \quad \text{// no deadlock}

  Thread 2 does not know about a

- \{ \text{next } a; \text{next } a; \} \text{ par } \text{next } a; \quad \text{// no deadlock}

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

Synchronizations on distinct variables may occur in any order

- \text{next } a; \text{ par } \text{next } b; \text{ par } \text{next } a; \text{ par } \text{next } b; \text{ par } \text{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
void f() {
    void tick; int i = 0;
    try {
        next tick; throw T;
    } par {
        while(true) { i = i + 1; next tick; }
    } catch(T) { i = i * 3; }
}

void g() {
    void tick; int i = 0;
    try {
        next tick; throw T;
    } par {
        while(true) { i = i + 1; next tick; }
    } catch(T) { i = i * 3; }
}

// i = 6

Exceptions

// thread 1
// thread 2
// runs forever
// never returns
void assoc(int key, Tree tree, \textbf{void} tick, \textbf{int} \&value) {
   if (tree == null) return;
   if (key == tree.key) {
      value = tree.value;
      \textbf{throw} \text{ Found};
   }
   \textbf{next} tick;
   assoc(key, tree.left, tick, value);
   \textbf{par} // possible race
   assoc(key, tree.right, tick, value);
}

\textbf{Not legal in SHIM}

\begin{itemize}
   \item a variable can be passed by reference \textbf{at most once} in a par
   \item a variable can be passed by value without restriction in a par
\end{itemize}
Inference Rules

```c
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
⇒ no shared memory ⇒ 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

```c
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; }
    }
}

⇒ The topmost, leftmost key occurrence has priority
```
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
void main() {
    int a = 0, b = 0;
    { // thread 1: rval: a, b
        next a; // a is 1, b is 0
        par { // thread 1a: rval a
            next b; // a is 0, b is 2
        } // thread 1b: rval b
        // a is 1, b is 2
    } par { // thread 2: lval: a, b
        next b = 2;
        next a = 1;
    } }
}

Communication
Extending the FIFO

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

{next a=3;a=5;} par {next a;b=a+1;} a=b;

\[
\begin{align*}
\text{next a=3;a=5;} & \quad \text{next a;b=a+1;} \\
\lambda \quad & \rightarrow \quad \text{a=b;} \\
\lambda, b: \mu / \lambda: 1, \mu: 2 & \quad \lambda, b: \mu / \lambda: 1, \mu: 2, \nu: 1 \\
\text{send a;a=5} & \quad \text{next a;b=a+1;} \\
\lambda \quad & \rightarrow \quad \text{a=b;} \\
\lambda, b: \mu / \lambda: 3, \mu: 2, \nu: 1 & \quad \lambda, b: \mu / \lambda: 3, \mu: 2, \nu: 3 \\
b=a+1; & \quad 0 \\
\lambda, b: \mu / \lambda: 5, \mu: 2, \nu: 3 & \quad \lambda, b: \mu / \lambda: 5, \mu: 4, \nu: 3 \\
a=b; & \quad \text{a=b;} \\
\lambda, b: \mu / \lambda: 5, \mu: 2, \nu: 3 & \quad \lambda, b: \mu / \lambda: 5, \mu: 4, \nu: 3 \\
a=b; & \quad \text{a=b;} \\
\lambda, b: \mu / \lambda: 5, \mu: 4 & \quad \lambda, b: \mu / \lambda: 4, \mu: 4 \\
\end{align*}
\]
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