R-SHIM: Deterministic Concurrency with Recursion and Shared Variables

SHIM

Olivier Tardieu and Stephen A. Edwards

Columbia University, New York, USA

Software/
Hardware
Integration
Medium

par and *next* added to standard imperative syntax

$e ::= L \mid V \mid op\ e \mid e\ op\ e \mid (\ e\ )$	**expressions**
$c ::= P(\ V(;V)^*\ )$	**procedure calls**
$s ::= V = e; \mid \{\ s^*\ \} \mid T\ V;$	**statements**
$\mid$`if (` e `)` s `else` $s \mid$ `while (` e `)` s	
$\mid c$ (`par` $c)^*$`;`	**concurrent procedure calls**
$\mid$`next` V `;`	**communication**
$d ::= T\ V \mid T\ \&V$	**parameter declarations**
$f ::=$ `void` $P(\ d(;d)^*\ )\ \{\ s^*\ \}$	**procedure declarations**
$p ::= f^*$ `void main() {` s^* `}`	**programs**

Basic syntax is an imperative language with two extensions:

- *par* for concurrency
- *next* for communication

The *next* construct reads or writes the next value of a "shared" variable

next A is a write if *A* is pass-by reference, a read otherwise

next is blocking: this is multi-way rendezvous communication

A process may only block on one variable at once, but *par* can be used to spawn a group of processes that block on a group of channels simultaneously

Communication with *next* is a blocking rendezvous operation

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

Processes are sequential, but *par* can be used to receive data in any order

```
void f(int &a, int &b, int &c) {
  a = 1;   b = 2;   c = 3;
  next a; next b; next c;
  a = 4;   b = 5;   c = 6;
  next b; next a; next c;
}
void receive(int &c) {
  next c;
}
void g(int a, int b, int c) {
  receive(a) par receive(b) par receive(c);
  // a,b,c = 1,2,3 here
  receive(a) par receive(b) par receive(c);
  // a,b,c = 4,5,6 here
}
void main() {
  int a; int b; int c;
  f(a,b,c) par g(a,b,c);
}
```

Pass-by-value variables must participate in the rendezvous

```
void f(int &c) {
   c = 1;    // modifies b
   next c;   // b <- a
             // c = b = 2 here
}
void g(int b) {
  f(b);
}
void h(int &a) {
  a = 2;     // same as top-level a
  next a;    // synchronize
}
void main() {
  int a;
  g(a) par h(a);
              // a = 2 here
}
```

A terminated process does not need to rendezvous

```
void f(int &a) {
  a = 1; next a;
  a = 2; next a;
}
void g(int a) {
  next a;   // a = 1 here
  next a;   // a = 2 here
}
void h(int a) {
  next a;   // a = 1 here
}
void main() {
  int a;
  f(a) par g(a) par h(a);
             // a = 2 here
}
```

Arguments may be passed by value or reference

By-value arguments are copied and may be modified independently

By-reference arguments allow a child to modify its parent's state

A variable may be passed by reference at most once per *par* block—this avoids race-inducing aliases

Pass-by-reference vs. pass-by-value arguments. By-value copy can be modified independently

```
void f(int  a) { a = a + 1; } // a passed by value
void g(int &a) { a = a + 5; } // a passed by reference
void main() {
  int a;
  a = 5;
  f(a) par g(a);   // invoke f & g concurrently
                   // a is always 10 here
}
```

May not pass a variable by reference twice
This prohibits race-inducing aliases

```
void f(int &a, int &b) {}
void g(int &c, int d) {}
void main() {
  int a; int b; int c; int d;
  f(a,b) par g(c,d);  // OK
  f(a,a) par g(c,d);  // No: a passed twice by reference
  f(a,b) par g(a,d);  // No: a passed twice by reference
  f(a,b) par g(c,a);  // OK
}
```

An *n*-place buffer using recursion

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

SHIM Guiding Principles

(Software/Hardware Integration Medium)

- Scheduling-independence
  Semantics of a program independent of the scheduler
  Scheduler makes nondeterministic choices; deterministic behavior results
- Safety
  Data races and other parallel pitfalls prohibited
  No pointers, or aliasing
- Simplicity
  Language has only two additional constructs over typical imperative language:
  - *par* for concurrency
  - *next* for communication

We have

- A formal semantics written in a structural operational style
- A prototype compiler that generates single-threaded C code
- A way of adding deterministic concurrent exceptions (upcoming Emsoft 2006 paper)

We plan

- A full, open-source compiler written in OCAML
- Support for arrays, structs, and fixed-precision arithmetic
- Synthesis for hardware, software, and mixed systems
- To take over the world of hardware/software codesign with SHIM