Concurrent Collections are Deterministic

Jens Palsberg

UCLA Computer Science Department
University of California, Los Angeles
palsberg@ucla.edu
This talk

- Determinism = deterministic input-output behavior
- Want: determinism by design; nondeterminism is a bug
- Theme: abstraction → determinism

- Goal 1: run-time check of determinism plus proof
  - Concurrent collections are deterministic: why and how

- Goal 2: compile-time check plus low annotation burden
  - Plasma simulation in X10: is load balancing deterministic?
**Analogy**

- Determinism in the 2010s = type checking in the 1980s

- The code was designed to satisfy the property
- The developers think the code satisfies the property
- Testing suggests the code satisfies the property
- The expert developers don’t want a new type system
- Average programmers need help!
Concurrent collections are deterministic

- Idea 1: dynamic single assignment
  - Or: forbid races

- Idea 2: blocking read
  - Or: forbid uninitialized-variable errors
Featherweight CnC syntax

- **Program:** \( p ::= f_i \ (\text{int} \ a) \ { \ d_i \ s_i } \ , \ i \in 1..n \)
- **Declaration:** \( d ::= n = \text{data.get}(e) ; d \)
  | \( \varepsilon \)
- **Statement:** \( s ::= \text{skip} \)
  | \( \text{if} \ (e>0) \ s_1 \ \text{else} \ s_2 \)
  | \( \text{data.put}(e_1,e_2) ; s \)
  | \( \text{prescribe} \ f_i(e) ; s \)
- **Expression:** \( e ::= ... \)
Featherweight CnC semantics (1/3)

A state in the semantics: \((A, T)\) or error

\[ T ::= T \parallel T \mid d \ s \]

\((A, \text{if } (e>0) \ s_1 \text{ else } s_2)\) \rightarrow (A, s_1) \quad \text{(if } \text{eval}(e) >0)\)

\((A, \text{if } (e>0) \ s_1 \text{ else } s_2)\) \rightarrow (A, s_2) \quad \text{(if } \text{eval}(e) \leq 0)\)

\((A, \text{prescribe } f_i(e); s)\) \rightarrow (A, (d_i, s_i)[a := \text{eval}(e)] \parallel s)\)
**Featherweight CnC semantics (2/3)**

\[
(A, \text{skip} \parallel T_2) \rightarrow (A, T_2) \quad (A, T_1 \parallel \text{skip}) \rightarrow (A, T_1)
\]

\[
(A, T_1) \rightarrow \text{error} \quad (A, T_2) \rightarrow \text{error}
\]

\[
(A, T_1 \parallel T_2) \rightarrow \text{error} \quad (A, T_1 \parallel T_2) \rightarrow \text{error}
\]

\[
(A, T_1) \rightarrow (A', T_1') \quad (A, T_2) \rightarrow (A', T_2')
\]

\[
(A, T_1 \parallel T_2) \rightarrow (A, T_1' \parallel T_2) \quad (A, T_1 \parallel T_2) \rightarrow (A, T_1 \parallel T_2')
\]
Traditional semantics

\[(A, \text{item.put}(e_1,e_2); s) \rightarrow (A[\text{eval}(e_1) := \text{eval}(e_2)]; s)\]

\[(A, n = \text{data.get}(e); d\ s) \rightarrow (A, (d\ s)[n := A(\text{eval}(e))])\]

if \(A(\text{eval}(e))\) is defined

\[(A, n = \text{data.get}(e); d\ s) \rightarrow \text{error, if } A(\text{eval}(e))\text{ is undefined}\]
CnC semantics (3/3)

\[(A, \text{item.put}(e_1,e_2); s) \rightarrow (A[\text{eval}(e_1) := \text{eval}(e_2)]; s)\]

(dynamic single assignment) if \(A(\text{eval}(e_1))\) is undefined

\[(A, \text{item.put}(e_1,e_2); s) \rightarrow \text{error}, \text{ if } A(\text{eval}(e_1))\) is defined

\[(A, n = \text{data.get}(e); d s) \rightarrow (A, (d s)[n := A(\text{eval}(e))])\]

(blocking read) if \(A(\text{eval}(e))\) is defined

\[(A, n = \text{data.get}(e); d s) \rightarrow \text{error}, \text{ if } A(\text{eval}(e))\) is undefined
Concurrent collections are deterministic

We use $\sigma$ to range over states.

A final state is one of:

- $(A, \text{skip})$
- error
- $(A, T)$ that hangs: all reads are blocked

Theorem: If $\sigma \rightarrow^* \sigma'$ and $\sigma \rightarrow^* \sigma''$ and $\sigma'$, $\sigma''$ are final states, then $\sigma' = \sigma''$. 
Proof of determinism

Lemma: (Confluence, Church-Rosser [1936])

If $\sigma \rightarrow^* \sigma'$ and $\sigma \rightarrow^* \sigma''$, then there exists $\sigma_c$ such that $\sigma' \rightarrow^* \sigma_c$ and $\sigma'' \rightarrow^* \sigma_c$
Switching gears to X10 now

- X10 = imperative language
  + async  // parallelism
  + finish  // synchronization

- Want: determinism without annotations
- Determinism = may-happen-in-parallel analysis
  + effect analysis
Plasma simulation in Fortran → X10

- Originally programmed in Fortran + MPI + pthreads
  by Viktor Decyk at UCLA

- Particle-in-cell computation

- Has run on many parallel hardware platforms over 20 years
  - 12 billion particles on 4,096 processors for months

- 3D version: 100,000 lines of Fortran

- 2D version: 10,000 lines of Fortran
  - 4,623 lines of X10
May-happen-in-parallel analysis [PPOPP 2010]

- For a program p,
  let MHP(p) be the true may-happen-in-parallel information

- **Theorem**: if |-- p : E, and E(main) = (M,O)
  then MHP(p) ⊆ M

- Type inference: given p, find E such that |-- p : E

- Type inference by constraint solving
MHP analysis of the plasma benchmark

- Plasma benchmark: 4,623 LOC; 151 async; 84 finish; 505 call
- 11,422 constraints
- About 16 seconds to solve the constraints (257 MB space)

#Pairs of async bodies that may happen in parallel:
- Self: 134 (in parallel with itself)
- Same: 120 (two different async bodies in the same method)
- Diff: 4 (two async bodies in different methods)
- Total: 258

- We didn’t find any false positives!
Plasma simulation does load balancing
Plasma simulation benchmark

class Particle { ... } ...

dist(:rank == 1) d = ...;

Particle[:rank == 1] a = new Particle[d](...);

while (...) { // 200,000 iterations
  ... a[i] = ... a[j] ... // 2/3 computation
  d = ...;
  a = new Particle[d](...a[i]...a[j]...); // 1/3 communication
}

Copyright @ 2009 UCLA
Conclusion

- Average programmers need determinism
- Determinism for CnC!
- Determinism for X10? Annotations?
- Challenge: load balancing for arrays of references