Æminium in a Nutshell

Sven Stork

May 5, 2010
Motivation/Hypothesis

Developers thinking sequential

- von Neuman architecture
- sequential programming languages
- programming is taught sequential
- natural limits of human brain

Software development process

- full program development $\Rightarrow$ module/component based approach
**Approach**

Relieve from sequential thinking

- move from *ordering* ⇒ *dependency* thinking
  - sequential ⇒ concurrent system
  - *Concurrency By Default*

Supporting current development styles

- design a (modular) system
  - composable
  - local reasoning/checking

**Target**

- High-level architectural concurrency/parallelism
Idea
use (access) permission to check and parallelize code

Approach
- track permission flow through program code (lexical ordering)
- check for correct usage of permission and data access (e.g. access shared data only within atomic block)
- use permission flow to infer data-dependency graph
Data Groups + Ownership

- *data groups* are (virtual) collections of objects
- every object belongs to exactly one data group (ownership)
- data groups are per object instance (form a hierarchy)
- use permission for control/check access to data group objects
exclusive only/unique permission

shared multiple permission to data group

protected shared data group protected by atomic block
exclusive: only/unique permission

share: exclusive \implies shared

shared: multiple permission to data group

atomic: shared \implies protected

protected: shared data group protected by atomic block
Example

```java
void main<exclusive α> () {
    Queue<α> q = new Queue<α>();
    Producer<α> p = new Producer<α>(q);
    Consumer<α> c = new Consumer<α>(q);

    // α : exclusive
    share <α> {
        // α : shared
        r.run();
    } // α : shared
    c.run();
} // α : exclusive
```
The KILLER System ???

NOPE. There more powerful systems out there but:

- powerful ≠ useful
- have an easy to understand abstraction
  - data groups/ownership encoded in types
  - no problems with aliasing
  - a simple “unpack” of internal groups
- prevent data-races
- can incrementally parallelize code (similar to Open MP)
- modularity + strong encapsulation approach seems to fit framework and library abstractions
Æminium

Current Status

• developed core calculus
• started work on correctness proof
• started to think about runtime implementation strategies

Open Issues

• how to handle I/O
• how to deal with legacy code
  • Can we implement Æminium in mainstream language?
• runtime implementation strategies
• granularity and load-balancing
• looking for a committee member from CMU