PART III PROJECT VIVA

INVESTIGATION INTO ALTERNATIVE PROGRAMMING ABSTRACTIONS USING "CAUSAL SYSTEMS"

Tristan Aubrey-Jones (taj105@ecs.soton.ac.uk)
School of Electronics and Computer Science, University of Southampton

Overview

- Project Description
  - Motivation
  - Objective & Approach
- Language Design
  - Language Model
  - Key Features
- Translator Implementation
- Demonstration
- Conclusion

Project Description

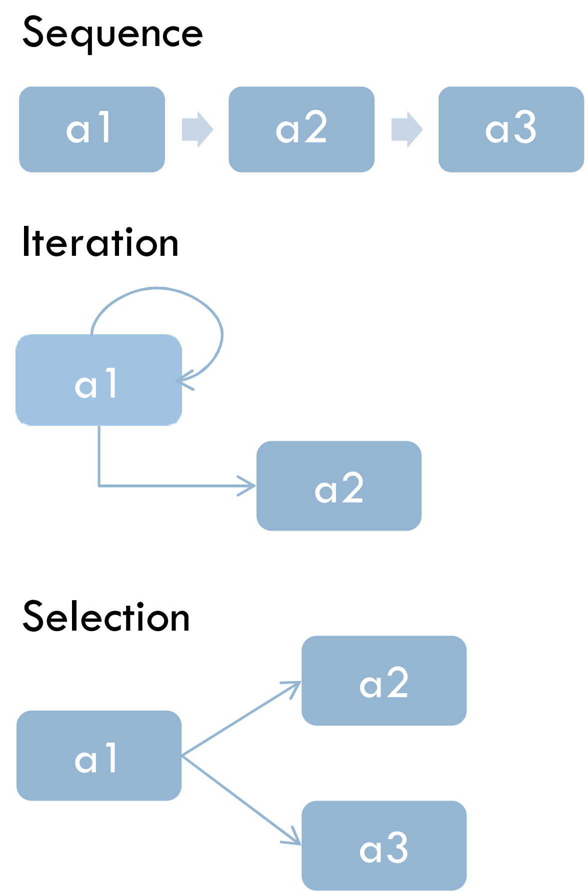
- Motivation
  - Architecture making a transition to parallel
    - In 2004 Intel scrapped 2 single-core processor designs, in favor of dual and quad-core designs.
  - Programs are no-longer just single threads
  - In 2006 a group from Berkley predicted
    - 1000's of cores per chip "many-core" architecture
    - Future programming models should be
      - More "human-centric"
      - Naturally parallel
      - Independent of number of processors.
  - Microsoft & Intel invest $20m parallel computing research

Project Description

- Objective (page 6)
  - Need an **implicitly parallel** programming language
    - To exploit new, and future hardware
    - So programs can scale to fully use available processors
    - That can easily run on distributed clusters, compute clouds
    - That will simplify concurrency
    - That are easy to learn and use

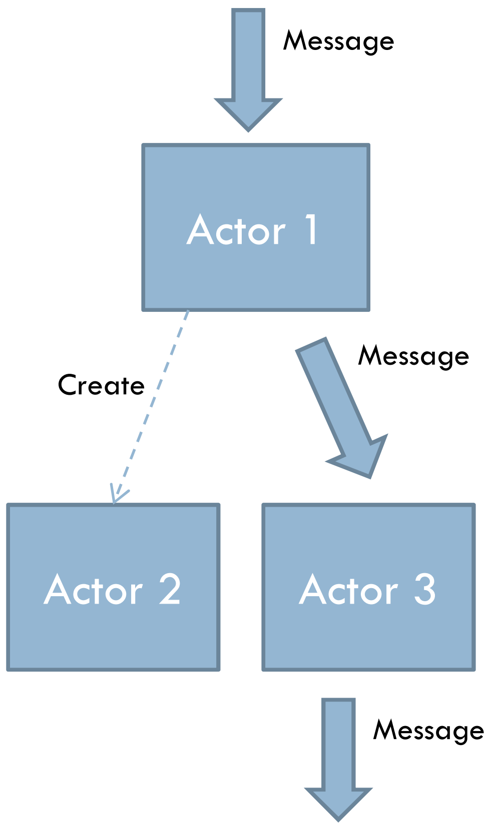
Project Description

- The Idea (page 7)
  - Base parallelism on causality
  - Objects **reacting** to events by sending events to other objects
  - All control structures can be described as patterns of message passing
  - Programs are "mini-universes": systems of interacting objects obeying rules governing their interaction.



Language Design

- The Actor Model of Computation (page 9)
  - Proposed by Hewitt 1973
  - Actors (objects) respond to messages by
    - Changing state
    - Sending more messages
    - Creating more actors
  - Benefits
    - Implicit concurrency
    - More powerful than functional or data flow
    - Object oriented
  - Drawbacks
    - Message passing inefficient
    - Shared resources must be actors



Language Design

- Approach
  - Build on Actor Model
  - Extend existing object oriented language (Java)
  - Linear typing
    - Linear objects only referenced by 1 identifier at a time
    - Use transference operator to move reference between identifier, and actors
    - Objects can be shared without synchronization constructs
    - Objects passed by reference on shared memory machines
    - Overloads existing message passing metaphor
  - Minimal Language & Extended Language

Language Design: Minimal

- The Actor Class (page 11-14)

```
public aclass PhilosopherActor extends Actor implements ForkConsumer {

    private TableActor table;
    private Fork left, right;
    private int state;

    public PhilosopherActor() {
        state = THINKING;
    }

    react (ForkPair forks) {
        ...
    }

    public static class AmHungry {
        ...
    }

    void becomeHungry() {
        ...
    }

}
```

Fields store state

Inheritance and Interfaces like object classes

Reactors define message handlers

Nested message type

Internal methods

Language Design: Minimal

- Reactor members (page 15)

```
public aclass PhilosopherActor extends Actor {

    private TableActor table;
    private Fork left, right;
    private int state;

    react (ForkPair forks) {
        if (state == HUNGRY) {

            left <-- forks.left;
            right <-- forks.right;

            state = EATING;
            eat();
        }

        table <-- new ForkPair(left, right);
        state = THINKING;
    }

}
```

Defined for a given message type

“Transfer” linear objects to fields

Change state

Send message

Create new linear object (destructively reading linear fields)

Language Design: Extended

- Expression Actors

Has a return type

```
public aclass Sorter returns int[] {

    react (int[] array) {
        ...
        return array;
    }

}
```

All reactors return values

```
Sorter sort = new Sorter();

int[] array = sort(new int[] {2, 3, 1});
```

Invoked synchronously like a function

  - Request/Response Pattern
  - Like functional programming “closures”

Language Design: Extended

- Fork Blocks (page 17)

```
Sorter sort1 = new Sorter();
Sorter sort2 = new Sorter();

fork (left = sort1(left);
      right = sort2(right);)
{
    array = merge(left, right);

    print("done.");
    return array;
}

print("sorting...");
```

Concurrent invocation

Continuation executes when all invocations complete

Continues immediately

  - Concurrent expression actor evaluation
  - Common programming pattern
  - Like asynchronous method call with call-back function

Language Design: Extended

- Further extensions

```
fork (left = sort1(left);
      right = sort2(right));
```

When body omitted, continues sequentially

```
react-when (counter > 0) {
    do();
    counter--;
}
```

Conditional reactors, execute while condition is true.

```
message IntPair(int a, int b);
```

Message object shorthand

```
event ClickHander onClick;
```

Actors can subscribe to actor events

Translator Implementation

- Translate into Java (page 26-29)
  - Tokenise using JFlex
  - Parse using CUP
  - Performs contextual analysis of AST
  - Translates extended constructs to minimal
  - Translates minimal constructs to Java
  - Emits Java code

Demonstration

- Dining Philosophers Problem (page 30)
  - Linear Types
  - Resource sharing using message passing
- Quicksort (page 31)
  - Recursive actor creation
  - Fork construct
- Calculator (page 32)
  - Event based programming, natural modularity
  - Design programs more like machines, with components

Conclusion

- Hybrid: Actor Model & Linear Types → New model for concurrency, with no need for synchronization constructs
  - Easier to understand (just one metaphor: "transference")
  - Impossible for accidental interference as no shared variables
  - Better performance: pass by reference
- Implicit parallelism + Familiar object oriented notation
- Clear interfaces via reactor members, rather than "receive" statements
- Scales to make use of available processors, and could be ported to run on clusters

Conclusion

- Successfully:
  - Created a new programming model
  - Developed an “implicitly parallel” language
  - Implemented a prototype compiler
  - Written and run programs to evaluate its features

Further Work

- Different return types for each reactor in expression actors
- Full semantic checking in translator
- Investigate "proximity"
- Code optimization & "auto-tuning"
- Deployment on open distributed systems

Questions?

The End