The Caltrop Actor Language

a short introduction

Johan Eker, Jörn W. Janneck
Chris Chang, Lars Wernli

The Ptolemy Group
UC Berkeley

December 14, 2001

What is Caltrop?

Caltrop is a (textual) language for writing dataflow actors.

- It compiles (one day) against the Ptolemy API (Pt/Java).
- ... but is intended to be retargetable, and usable in a wide variety of contexts.
- It is designed as a *domain-specific language*.

Can you guess what this does?

```
actor A (Double k)
    Double Input1, Double Input2 ==> Double Output:

    action [a], [b] ==> [k*(a + b)] end
end
```

Or this?

```
actor B ()
    Double Input ==> Double Output:

    Integer n := 0;
    Double sum := 0;

    action [a] ==> [sum / n] :
        n := n + 1;
        sum := sum + a;
    end
end
```

Motivation

Writing simple actors should be simple.

- But: The Ptolemy API is very rich, and writing actors in it requires considerable skill.
- However: Many Ptolemy actors have considerable commonalities - they are written in a stylized format.

Motivation

We should **generate** many actors from a more abstract description.

- Benefits:
  - reduces amount of code to be written
  - makes writing actors more accessible
  - reduces error probability
  - makes code more versatile
    - actors may be retargeted to other platforms, or new versions of the Ptolemy API

Why is Caltrop useful for me?

- For Ptolemy users:
  - Makes it easier to write atomic actors.
  - Makes Ptolemy accessible to a wider audience.
- For Ptolemy ‘hackers’:
  - Reduces possibilities for bugs.
  - Makes actors more reusable.
  - Enables analysis and efficient code generation.

Some language features

Multiple actions, action conditions

```
actor C ()
    Double Input ==> Double Output:

 action [a] ==> [a] where a >= 0 end

 action [a] ==> [-a] where a < 0 end
end
```

```
actor D ()
    Double Input ==> Double Output:

 action [a] ==> [abs(a)] end
end
```

Nondeterminism

```
actor E ()
    Double Input ==> Double Output:

 action [a] ==> [a] end

 action [a] ==> [-a] end
end
```

- More than one action may be firable.
- Caltrop does not specify how this is resolved.
- It allows deterministic as well as non-deterministic implementations.
- Often, determinism can be ascertained statically.

Port patterns

```
actor PairwiseSwap [T] ()
    T Input  ==> T Output:

    action [a, b] ==> [b, a] end
end
```

- examples
  - **[a, b, c]**
  - **[a, b, c | s]**
  - **[| s]**

Repeated patterns

```
actor ReversePrefix [T] (Integer n)
    T Input  ==> T Output:

    action [a] repeat n ==> [reverse(a)] repeat n end
end
```

Channel selectors

```
actor Switch [T] ()
    multi T Data, Integer Select  ==> T Output:

    action [a] i, [i] ==> [a] end
end
```

Port tags

```
actor Switch [T] ()
    multi T Data, Integer Select  ==> T Output:

    action Data:: [a] i, Select:: [i] ==> [a] end
end
```

```
actor Switch [T] ()
    multi T Data, Integer Select  ==> T Output:

    action Select:: [i], Data:: [a] i ==> [a] end
end
```

Action tags, action selectors

```
actor FairMerge [T] ()
     T Input1, T Input2  ==> T Output:

     A: action Input1:: [a] ==> [a] end

     B: action Input2:: [a] ==> [a] end

     selector
          (AB)*
     end
end
```

other selectors are conceivable, e.g.

- **(AB)* | (BA)***
- **((AB) | (BA))***

Action conditions, state

```
actor FairMerge [T] ()
    T Input1, T Input2  ==> T Output:

    Integer s := 1;

    action Input1:: [a] ==> [a] where s = 1:
        s := 2;
    end

    action Input2:: [a] ==> [a] where s = 2:
        s := 1;
    end
end
```

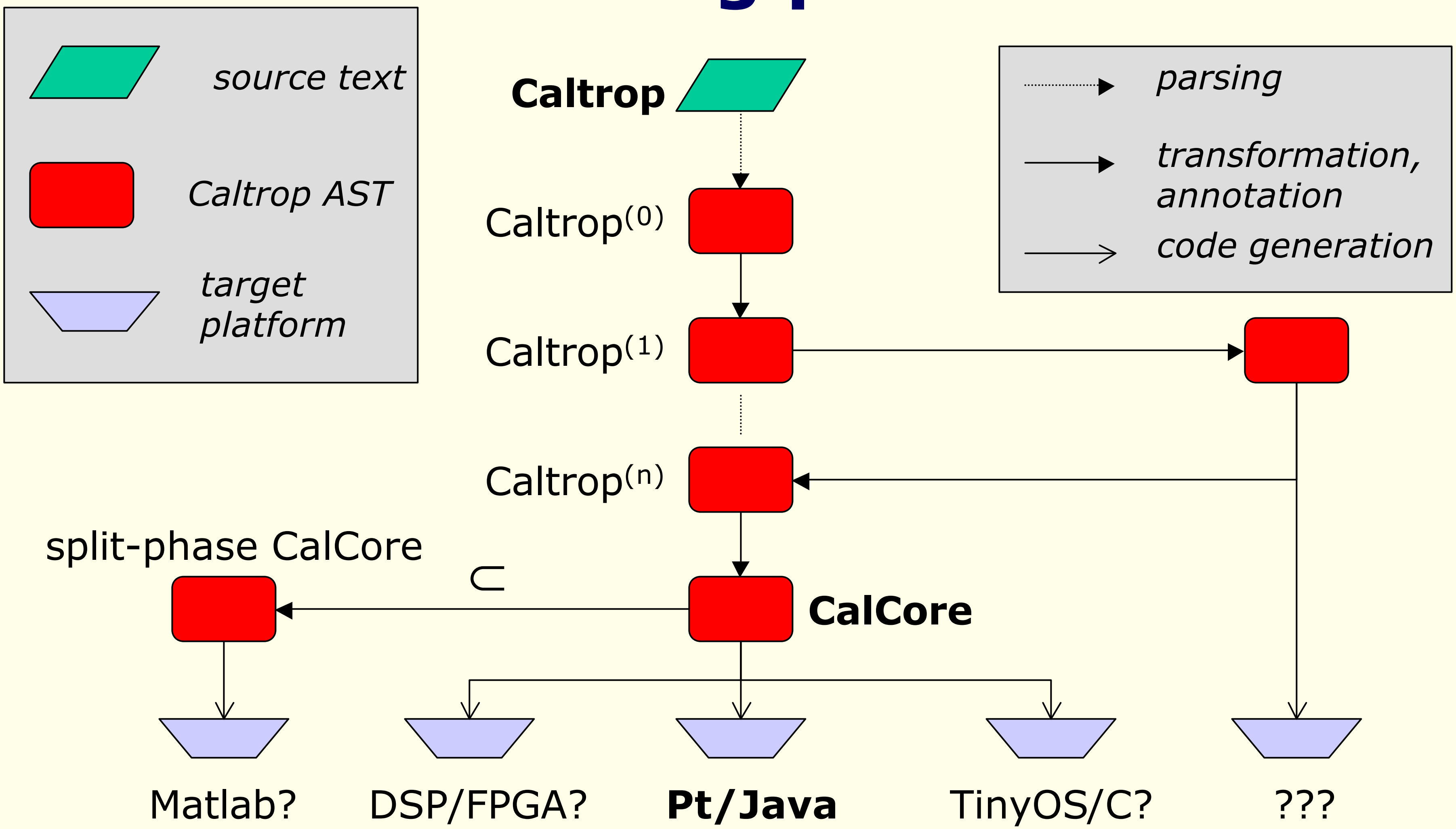
Overview of the implementation

general architecture

aspects of the Pt/Java implementation

Caltrop implementation

—the big picture.



Caltrop implementation

- many small modules
  - transformers, annotaters, checkers
- transforming Caltrop into some language subset
- **CalCore**
  - small, semantically complete subset
  - minimal language for code generation, analysis, transformation
  - built on functional and procedural closures

Transformers & annotators

- replace control structures
- replace port patterns
- sort variable declarations
- replace operators
- propagate type information
- check static properties
- tag port patterns
- ...

Caltrop implementation

- Benefits:
  - Much of the ‘hard’ stuff can be done on the same data structure, exploiting the regularities of the Caltrop/CalCore semantics and the existing software infrastructure.
  - Code generators becomes relatively small, thus making retargeting easier.
  - Intermediate results are valid Caltrop programs, making debugging a lot easier.
    - We can look at them easily.
    - We can use the parser and the well-formedness/type checkers themselves as debugging tools!

Switch in CalCore*

```
actor Switch [T] ()
  multi T Data, Integer Select ==> T Output :

  action Select::[|G0], Data::[|G2] all ==> Output::[a]
    where G1, G3
    with
      Boolean G1 := available(G0, 1),
      Integer i := if G1 then G0[0] else null end,
      Integer G4 := i,
      Boolean G3 := available(G2[G4], 1),
      Integer a := if G3 then G2[G4][0] else null end
    end
end
```

***Actually, we are cheating here; CalCore is in fact even more primitive. And even less readable...**

CalCore ==> Pt/Java

- different programming models
  - single atomic action vs `prefire/fire`^{n}`/postfire`
  - referentially transparent access to channels vs. token consuming `read` methods
  - stateful computation vs state-invariant `fire`

CalCore ==> Pt/Java

- mapping Caltrop notions to Pt/Java concepts
  - parameters ==> attributes + attribute changes
  - types and type checking
  - ...

What goes into `prefire()`?

```
actor Switch [T] ()
  multi T Data, Integer Select ==> T Output :

  action Select::[|G0], Data::[|G2] all ==> Output::[a]
    where G1, G3
    with
      Boolean G1 := available(G0, 1),
      Integer i := if G1 then G0[0] else null end,
      Integer G4 := i,
      Boolean G3 := available(G2[G4], 1),
      Integer a := if G3 then G2[G4][0] else null end
    end
end
```

All computation that is required in order to decide firability?

Just the part of it before the first token access?

`prefire/fire`

- aggressive vs defensive condition evaluation
- incrementally computing conditions
- reusing computation done in `prefire`
- Should tokens read in `prefire` be kept if `prefire` returns `false`?

fire/postfire

```
action Input1:: [a] ==> [a]
  where s = 1 :

    s := 2;
end
```

```
action Input1::[|G0] ==> Output::[a]
  where equals(s,1), G1
  with
    Boolean G1 := available(G0, 1),
    T a := if G1 then G0[0] else null end :

    s := 2;
end
```

prefire

fire

postfire

Computation that does not affect the output can be done in postfire.

State management

If state needs to be changed in order to compute output, it needs to be shadowed.

- safe state management
  - referentially transparent expressions
  - no aliasing of stateful structures

State management

```
actor B ()
    Double Input ==> Double Output:

    Integer n := 0;
    Double sum := 0;

    action [a] ==> [sum / n] :
        n := n + 1;
        sum := sum + a;
    end
end
```

The division between fire and postfire can be expressed in Caltrop (btw, this is a non-CalCore transformation) using action tags and selectors.

State management

```
actor B ()
  Double Input ==> Double Output:

  Integer n := 0; Integer n$shadow;
  Double sum := 0; Double sum$shadow;

  fire: action [a] ==> [sum$shadow / n$shadow] :
    n$shadow := n; sum$shadow := sum;
    n$shadow := n$shadow + 1;
    sum$shadow := sum$shadow + a;
  end

  postfire: action ==> :
    n := n$shadow; sum := sum$shadow;
  end

  selector (fire+ postfire)* end
end
```

Things we need to do…

Short term “grunt” work ☹

- well-formedness checks
- type system, type checking
- split-phase analyses
- interpreter (+ wrapper for Pt/Java)
- optimizations, big and small
- other transformers, annotators

Fun mid-term projects ☺

- static analysis of actor properties (data rates, dependency on state, parameters, input, ...)
- relation to MoC (e.g. BDF)
- computing interface automata from actor descriptions
- code generators to other platforms and languages
- code generation for composite actors?

Even funner long-term projects ☺☺

- generic code generation framework
  - maybe based on Calif?
- extending the language
  - higher-order constructs
  - domains/directors+receivers
- a formal semantics, a framework for actor analysis

Caltrop resources

www.gigascale.org/caltrop

Meeting:

Tuesdays, 1:30pm,

DOP Center Library

(We need Caltroopers!)

Thanks.