Proposal: Concurrent KAT for Calyx

Jan-Paul Ramos-Dávila
Cornell University
Ithaca, NY, USA
jvr34@cornell.edu

Abstract

This proposal outlines a project to formalize and implement parallelism semantics in Calyx using ideas from Kleene Algebra with Tests (KAT) and Synchronous Kleene Algebra (SKA). The project aims to provide a rigorous foundation for reasoning about parallel execution in Calyx while addressing its specific needs for hardware design.

1 Project Outline

1.1 Background and Motivation

- Overview of Calyx and its current `par` semantics
- Need for formal semantics to reason about parallel execution
- Introduction to Kleene Algebra with Tests and Synchronous Kleene Algebra

1.2 Formalization of Calyx Parallelism using SKA

- Define a core subset of Calyx focusing on `par` constructs
- Map Calyx constructs to SKA operators
- Define semantics for `par` using SKA's synchronous product operator
- Prove key properties of `par` using SKA axioms

1.3 Extend SKA for Calyx-specific Needs

- Introduce latency-insensitive semantics to SKA
- Define a "loose" synchronous product for non-deterministic execution order
- Prove soundness and completeness of the extended algebra

1.4 Operational Semantics and Implementation

- Define small-step operational semantics for Calyx `par` based on the SKA model
- Implement a reference interpreter for the core Calyx subset
- Develop compilation strategies for `par` based on the formal semantics

2 Operational Semantics

The operational semantics for Calyx's par construct can be defined using a small-step semantics approach. This aligns most closely with the "Explicit Synchronization" semantics, as it allows for explicit control over the execution order and synchronization points.

We define the following judgment for the execution of Calyx programs:

$$\frac{\langle p, \sigma \rangle \to \langle p', \sigma' \rangle}{\text{Program } p \text{ in state } \sigma \text{ steps to } p' \text{ with new state } \sigma'} \; \textsc{Step}$$

For the par construct, we define the following rules:

$$\frac{\langle p_1, \sigma \rangle \to \langle p_1', \sigma' \rangle}{\langle \mathsf{par}\{p_1, p_2\}, \sigma \rangle \to \langle \mathsf{par}\{p_1', p_2\}, \sigma' \rangle} \; \textsc{Par-Left}$$

$$\frac{\langle p_2, \sigma \rangle \to \langle p_2', \sigma' \rangle}{\langle \mathsf{par}\{p_1, p_2\}, \sigma \rangle \to \langle \mathsf{par}\{p_1, p_2'\}, \sigma' \rangle} \; \textsc{Par-Right}$$

$$\frac{p_1 = \mathsf{done} \quad p_2 = \mathsf{done}}{\langle \mathsf{par}\{p_1, p_2\}, \sigma \rangle \to \langle \mathsf{done}, \sigma \rangle} \; \textsc{Par-Done}$$

For groups, we can define:

$$\frac{g \text{ is a group}}{\langle g, \sigma \rangle \to \langle g_{\text{body}}, \sigma[g.\mathsf{go} \mapsto 1] \rangle} \; \textsc{Group-Start}$$

$$\frac{g.\mathsf{done} = 1 \text{ in } \sigma}{\langle g_{\text{body}}, \sigma \rangle \to \langle \mathsf{done}, \sigma \rangle} \; \textsc{Group-Done}$$

To handle the latency-insensitive nature of Calyx, we can introduce a rule for delays:

$$\frac{\text{true}}{\langle \delta, \sigma \rangle \to \langle \mathsf{done}, \sigma \rangle} \; \textsc{Delay}$$

This rule allows for arbitrary delays between operations, reflecting the latency-insensitive design of Calyx.

These operational semantics reflect the "Explicit Synchronization" approach in the following ways:

- The Par-Left and Par-Right rules allow for non-deterministic interleaving of parallel executions, capturing the idea that parallel arms can execute in any order.
- The Group-Start and Group-Done rules explicitly model the activation and completion of groups, similar to the proposed synchronization points.
- The Delay rule allows for modeling of latency-insensitive designs, where the exact timing of operations is not fixed.

3 Technical Questions

3.1 Questions for Synchronous Kleene Algebra Authors

1. How can the SKA framework be extended to handle latency-insensitive designs?
2. Is it possible to define a "loose" synchronous product that allows for non-deterministic execution order while preserving essential properties?
3. How can we incorporate the concept of clock cycles or timing into the SKA framework to better model hardware behavior?
4. Are there any established techniques for handling state and side effects in SKA that could be applicable to Calyx's stateful components?

3.2 Questions for Calyx Semantics

1. How can we formally define the semantics of Calyx's groups and their interaction with the `par` construct?
2. What is the precise meaning of the "done" signal in Calyx, and how should it be incorporated into the formal semantics?
3. How should we handle potential conflicts between parallel executions of groups that modify the same state?
4. What is the desired level of determinism for Calyx's `par` construct, and how can this be formally specified?

4 KA Formulas in Calyx Context

To illustrate how KA formulas would look in the context of a Calyx program, consider the following Calyx snippet:

```
group g1 {
  x.in = a.out;
  g1[done] = x.done;
}

group g2 {
  y.in = b.out;
  g2[done] = y.done;
}

control {
  par { g1; g2 }
}
```

We can represent this using KA formulas as follows:

$$\begin{aligned} g1 &:= (x.in \leftarrow a.out) \cdot (g1[done] \leftarrow x.done) \\ g2 &:= (y.in \leftarrow b.out) \cdot (g2[done] \leftarrow y.done) \\ par(g1, g2) &:= g1 \times g2 \end{aligned}$$

Where $\times$ represents the synchronous product from SKA. We can then define properties such as:

$$par(g1, g2) = par(g2, g1) \quad \text{(commutativity)}$$

$$par(g1, par(g2, g3)) = par(par(g1, g2), g3) \quad \text{(associativity)}$$

To handle latency-insensitive designs, we might introduce a delay operator δ and modify our formulas:

$$\begin{aligned} g1 &:= (x.in \leftarrow a.out) \cdot \delta \cdot (g1[done] \leftarrow x.done) \\ g2 &:= (y.in \leftarrow b.out) \cdot \delta \cdot (g2[done] \leftarrow y.done) \end{aligned}$$

This allows for a variable number of cycles between the input assignment and the done signal.