Concurrent Kleene Algebra with Tests for Calyx Parallelism

Jan-Paul Ramos

September 16, 2024

1 Introduction

Calyx is an intermediate language for compiling high-level descriptions of hardware accelerators into low-level hardware descriptions. It uses a split representation that separates structural descriptions from control flow. However, the current semantics of parallelism in Calyx, particularly the `par` construct, present several challenges for analysis and optimization.

2 Problems with Current Par Semantics

The main issues with the current `par` semantics in Calyx are:

1. Undefined behavior for data races
2. Non-determinism in execution order
3. Lack of formal semantics for parallel constructs
4. Difficulty in expressing complex parallel patterns
5. Compilation challenges in preserving parallelism
6. Inaccurate dataflow analysis, particularly for liveness

3 The Par CFG Problem

Traditional control flow graphs (CFGs) and dataflow analyses are not well-suited for handling parallel execution, which is a key feature of Calyx's `par` construct. This leads to challenges in accurately representing and analyzing parallel code.

3.1 Example

Consider the following Calyx code:

```
F; // wr x
par {
  A; // wr x
  B;
  C; // rd x
}
G; // rd x
```

The key question is: Is `x` alive between `F` and the beginning of `par`?

3.2 Challenges

1. Traditional CFGs assume sequential execution, but `par` allows simultaneous execution of multiple statements.
2. Simple union or intersection of liveness sets from parallel branches leads to incorrect results.
3. Information flow between siblings in a `par` statement is complex and can affect optimization opportunities.

3.3 Proposed Solution

To address these issues, Calyx proposes passing `gen` and `kill` sets along with `live` sets. The liveness after a `par` block can be computed as:

$$(\bigcup \text{live(children)} - \bigcup \text{kill(children)}) \cup \bigcup \text{gen(children)}$$

This approach allows for more accurate liveness analysis in the presence of parallel execution.

3.4 Implications

- In Calyx, we must consider `x` as live at `B` within the `par` block, as siblings may interact arbitrarily.
- This conservative approach limits some optimization opportunities but ensures correctness in the presence of potential inter-sibling interactions.
- The complexity of parallel dataflow analysis highlights the need for a more robust formal framework, such as CKAT, to reason about parallel execution in Calyx.

4 Standard Operators in Kleene Algebra with Tests

Kleene Algebra with Tests (KAT) extends Kleene Algebra with boolean tests, providing a powerful framework for reasoning about programs. The standard operators in KAT and their significance for program analysis are as follows:

4.1 Basic Operators

- **Addition (+):**
  - Represents alternative paths or nondeterministic choice.
  - In CFGs: models branching or multiple execution paths.
  - Example: $a + b$ means "execute a or execute b".
- **Multiplication ($\cdot$):**
  - Represents sequential composition.
  - In CFGs: models the sequence of operations.
  - Example: $a \cdot b$ means "execute a then execute b".
- **Kleene star (*):**
  - Represents iteration (zero or more times).
  - In CFGs: models loops and repetition.
  - Example: a^* means "execute a zero or more times".
- **Zero (0):**
  - Represents the empty set or failed computation.
  - In analysis: often used to represent infeasible paths.
- **One (1):**
  - Represents the empty string or identity function.
  - In analysis: often used to represent a no-op or skip.

4.2 Test Operators

- **Complementation ($\neg$):**
  - Represents the negation of a test.
  - In analysis: used to represent conditions and their negations.
  - Example: If t tests for "$x > 0$", then $\neg t$ tests for "$x \leq 0$".
- **Meet ($\sqcap$) and Join ($\sqcup$):**
  - Represent conjunction and disjunction of tests.
  - In analysis: used to combine conditions.
  - Example: $t_1 \sqcap t_2$ means "both t_1 and t_2 hold".

4.3 Importance in Program Analysis

These operators are crucial for program analysis for several reasons:

1. **Expressiveness:** They allow us to represent complex program structures and behaviors in a concise algebraic form.
2. **Compositionality:** We can build representations of large programs by composing smaller parts, mirroring the structure of the program itself.
3. **Equational Reasoning:** KAT provides a rich set of equational axioms that allow us to manipulate and reason about program representations algebraically.
4. **Verification:** We can use KAT to prove program equivalence and correctness properties.
5. **Optimization:** The algebraic framework allows us to justify program transformations and optimizations formally.

4.4 Application to Calyx and Parallel Composition

In the context of Calyx and parallel composition, these operators allow us to:

- Represent sequential and parallel compositions of program fragments.
- Express complex control flow involving both sequential and parallel execution.
- Reason about the interaction between tests (e.g., liveness predicates) and program actions.
- Formulate and prove properties of parallel executions, such as non-interference or data race freedom.

For example, we can represent a Calyx `par` construct as:

$$\texttt{par \{ A; B \}} = A \| B$$

Where $\|$ is our additional parallel composition operator. We can then use the standard KAT operators to reason about the behavior of this parallel composition in the context of the larger program:

$$(p \cdot (A \| B) \cdot q)^*$$

This expression represents a loop where p is a test, followed by the parallel execution of A and B, followed by another test q. The Kleene star allows this sequence to repeat.

5 Proposed Solution: Concurrent Kleene Algebra with Tests (CKAT)

We propose using an extended version of Kleene Algebra with Tests, which we call Concurrent KAT (CKAT), to formalize the semantics of Calyx's `par` construct.

5.1 Key Components of CKAT

- Standard KAT operators: $+, \cdot, *, 0, 1$
- Parallel composition operator: $\|$
- Atomic action predicates: $[a]$ (represents an atomic action 'a')
- Conflict relation: # (represents conflicting actions)

5.2 Axioms for CKAT

In addition to standard KAT axioms, we add:

$$\begin{aligned} a\|b &= b\|a \text{ (Commutativity)} \\ (a\|b)\|c &= a\|(b\|c) \text{ (Associativity)} \\ (a+b)\|c &= (a\|c) + (b\|c) \text{ (Distributivity)} \\ [a]\|[b] &= [a]\cdot[b] + [b]\cdot[a] \text{ (If } a\#b\text{, Conflict resolution)} \end{aligned}$$

5.3 Representing Calyx Par Constructs

We can represent a Calyx `par` block as:

$$\texttt{par \{ A; B; C \}} = A\|B\|C$$

Where A, B, and C are CKAT expressions representing the behavior of each parallel branch.

6 Implementation in Calyx

To implement CKAT in Calyx, we need to modify several parts of the codebase:

1. Extend the control flow representation in `control_order.rs`
2. Modify dataflow analysis in `dataflow_order.rs`
3. Update live range analysis in `live_range_analysis.rs`
4. Adapt optimization passes in `compaction_analysis.rs`
5. Update code generation in `static_schedule.rs`

7 Benefits of CKAT Framework

The CKAT framework provides several benefits for Calyx:

- Formal semantics for parallel execution
- Precise reasoning about data races and conflicts
- Improved dataflow analysis, especially for liveness in parallel constructs
- Foundation for proving correctness of optimizations
- Ability to express and analyze complex parallel patterns
- Better preservation of parallelism during compilation

8 Example: Liveness Analysis with CKAT

Consider the following Calyx code:

```
F; // wr x
par {
  A; // wr x
  B;
  C; // rd x
}
G; // rd x
```

We can represent this in CKAT as:

$$(wr_x \cdot F) \cdot ((wr_x \cdot A) \| B \| (rd_x \cdot C)) \cdot (rd_x \cdot G)$$

Using CKAT, we can define axioms that capture:

1. A write followed by a parallel composition that includes another write kills the first write.
2. A read in any branch of a parallel composition makes the variable live for the entire `par` block.

This allows us to correctly analyze that x is not live between F and the `par` block, but is live within and after the `par` block.

9 CKAT Representation for Improved CFGs

9.1 CKAT-based Control Flow Graphs

CKAT allows us to represent parallel execution more accurately in control flow graphs by introducing a parallel composition operator ($\|$) alongside the traditional sequential composition ($\cdot$). This enables us to construct CFGs that explicitly represent parallel execution paths.

9.2 Tests in CKAT for Calyx

Tests are predicates that can be used to represent conditions or state. For our Calyx example, we can use tests to represent the liveness of variables. Let's define the following tests:

- l_x: Test that evaluates to true if x is live
- w_x: Test that evaluates to true if x is written
- r_x: Test that evaluates to true if x is read

9.3 CKAT Formula for the Example

Let's represent our example using CKAT:

$$(w_x \cdot F) \cdot ((w_x \cdot A) \| B \| (r_x \cdot C)) \cdot (r_x \cdot G)$$

Intuitively, this formula represents:

- $w_x \cdot F$: Write to x in F
- $(w_x \cdot A) \| B \| (r_x \cdot C)$: Parallel execution of A (writing x), B, and C (reading x)
- $r_x \cdot G$: Read from x in G

9.4 Liveness Analysis with CKAT

To perform liveness analysis, we can define axioms in CKAT:

$$\begin{aligned} w_x \cdot l_x &= 0 \text{ (a write kills liveness)} \\ r_x &\leq l_x \text{ (a read implies liveness)} \\ (a \| b) \cdot l_x &= (a \cdot l_x) \| (b \cdot l_x) \text{ (liveness distributes over parallel comp.)} \end{aligned}$$

Using these axioms, we can reason about liveness in our example:

$$\begin{aligned} &(w_x \cdot F) \cdot ((w_x \cdot A) \| B \| (r_x \cdot C)) \cdot (r_x \cdot G) \cdot l_x \\ &= (w_x \cdot F) \cdot ((w_x \cdot A \cdot l_x) \| (B \cdot l_x) \| (r_x \cdot C \cdot l_x)) \cdot (r_x \cdot G) \\ &= (w_x \cdot F) \cdot (0 \| (B \cdot l_x) \| (r_x \cdot C)) \cdot (r_x \cdot G) \\ &= (w_x \cdot F) \cdot (B \cdot l_x \| r_x \cdot C) \cdot (r_x \cdot G) \\ &= (w_x \cdot F) \cdot r_x \cdot C \cdot (r_x \cdot G) \end{aligned}$$

This derivation shows that x is not live after F (due to the write), becomes live in the `par` block due to the read in C, and remains live through G.

9.5 Implementation in Calyx

To implement this CKAT-based analysis in Calyx, we would need to extend the current CFG representation and dataflow analysis. Here's a sketch of how this might look in Rust:

```
enum CKATExpr {
    Seq(Vec<CKATExpr>),
    Par(Vec<CKATExpr>),
    Write(String),   // e.g., Write(red"redxred")
    Read(String),    // e.g., Read(red"redxred")
    Test(String),    // e.g., Test(red"redl_xred")
}

bluestruct CKATAnalysis {
    expr: CKATExpr,
}

impl CKATAnalysis {
    fn analyze_liveness(&self) → HashSet<String> {
        bluelet mut live_vars = HashSet::new();
        self.analyze_expr(&self.expr, &mut live_vars);
        live_vars
    }

    fn analyze_expr(&self, expr: &CKATExpr, live_vars: &mut HashSet<String>)
        bluematch expr {
            CKATExpr::Seq(exprs) ⇒ {
                bluefor expr bluein exprs.iter().rev() {
                    self.analyze_expr(expr, live_vars);
                }
            }
            CKATExpr::Par(exprs) ⇒ {
                bluelet mut par_live = HashSet::new();
                bluefor expr bluein exprs {
                    bluelet mut branch_live = live_vars.clone();
                    self.analyze_expr(expr, &mut branch_live);
                    par_live.extend(branch_live);
                }
                *live_vars = par_live;
            }
            CKATExpr::Write(var) ⇒ {
                live_vars.remove(var);
            }
            CKATExpr::Read(var) ⇒ {
                live_vars.insert(var.clone());
            }
```

```
                CKATExpr::Test(_) => {} // Tests don't affect liveness directly
            }
        }
}
```

This implementation sketch shows how we might represent CKAT expressions and perform liveness analysis based on these expressions. The `analyze_liveness` function would traverse the CKAT expression, updating the set of live variables according to the CKAT axioms we defined earlier.

9.6 CKAT Expression Representation

```
enum CKATExpr {
    Seq(Vec<CKATExpr>),
    Par(Vec<CKATExpr>),
    Write(String),
    Read(String),
    Test(String),
}
```

This enum represents the core elements of CKAT:

- `Seq`: Represents sequential composition ($\cdot$ in CKAT)
- `Par`: Represents parallel composition ($\|$ in CKAT)
- `Write` and `Read`: Represent atomic actions
- `Test`: Represents tests in CKAT

These constructs allow us to build expressions that mirror the structure of Calyx programs, including parallel constructs.

9.7 CKAT Analysis

```
struct CKATAnalysis {
    expr: CKATExpr,
}

impl CKATAnalysis {
    fn analyze_liveness(&self) -> HashSet<String> {
        let mut live_vars = HashSet::new();
        self.analyze_expr(&self.expr, &mut live_vars);
        live_vars
    }
    // ...
}
```

This structure encapsulates a CKAT expression and provides methods to analyze it. The `analyze_liveness` function performs a liveness analysis on the CKAT expression, which relates to how we use CKAT to reason about program properties.

9.8 Expression Analysis

```
fn analyze_expr(&self, expr: &CKATExpr, live_vars: &mut HashSet<String>) {
    match expr {
        CKATExpr::Seq(exprs) => {
            for expr in exprs.iter().rev() {
                self.analyze_expr(expr, live_vars);
            }
        },
        CKATExpr::Par(exprs) => {
            let mut par_live = HashSet::new();
            for expr in exprs {
                let mut branch_live = live_vars.clone();
                self.analyze_expr(expr, &mut branch_live);
                par_live.extend(branch_live);
            }
            *live_vars = par_live;
        },
        CKATExpr::Write(var) => {
            live_vars.remove(var);
        },
        CKATExpr::Read(var) => {
            live_vars.insert(var.clone());
        },
        CKATExpr::Test(_) => {}
    }
}
```

This function is where the CKAT theory is applied in practice:

- **Sequential Composition**: For `Seq`, we analyze expressions in reverse order, mimicking how liveness propagates backwards in a program. This corresponds to the CKAT axiom $a \cdot b \cdot l_x = a \cdot (b \cdot l_x)$.
- **Parallel Composition**: For `Par`, we analyze each branch independently and then combine the results. This relates to the CKAT axiom $(a \| b) \cdot l_x = (a \cdot l_x) \| (b \cdot l_x)$.
- **Write**: A write operation removes a variable from the live set, corresponding to the CKAT axiom $w_x \cdot l_x = 0$.
- **Read**: A read operation adds a variable to the live set, corresponding to the CKAT axiom $r_x \leq l_x$.

- **Test**: Tests don't directly affect liveness in this simple model, but in a more complex analysis, they could be used to represent conditions that affect liveness.

9.9 Connecting Theory to Implementation

The implementation translates CKAT concepts into practical analysis:

1. **Algebraic Structure**: The `CKATExpr` enum mirrors the algebraic structure of CKAT, allowing us to represent complex programs with both sequential and parallel composition.
2. **Compositional Analysis**: The recursive nature of `analyze_expr` reflects the compositional nature of CKAT, where properties of complex expressions are derived from properties of their components.
3. **Parallel Semantics**: The handling of `Par` expressions captures the essence of how CKAT deals with parallel composition, considering all possible interleavings implicitly.
4. **Action Semantics**: The treatment of `Read` and `Write` operations corresponds to how CKAT axioms describe the effect of these actions on liveness.

9.10 Limitations and Future Enhancements

While this implementation captures the essence of CKAT-based analysis, it has limitations:

- It doesn't fully capture the algebraic laws of CKAT, such as distributivity or associativity.
- The handling of tests is simplistic and could be extended to incorporate more complex boolean algebra.
- It doesn't include mechanisms for proving program equivalence or applying CKAT-based optimizations.

Future enhancements could include:

- Implementing a full CKAT algebra with all operators and laws.
- Extending the analysis to handle more complex properties beyond simple liveness.
- Incorporating techniques for program equivalence proofs and optimizations based on CKAT axioms.