Georgia Tech College of Computing

Featherweight X10

Florian Peter Donhauser
fdonhauser3@gatech.edu
Technical University of Munich

James Rohan Gangavarapu
james.gangavarapu@gatech.edu
Georgia Institute of Technology

Divyanshu Upreti
dupreti7@gatech.edu
Georgia Institute of Technology

ABSTRACT

We take up the language Featherweight X10, a minimal subset of the X10 language, and formalize its syntax and semantics in the proof assistant Coq. We go on to prove the progress theorem for FX10 in Coq and thereby conclude that the language is deadlock free.

KEYWORDS

X10, Coq, syntax, semantics, progress proof, deadlock freedom

1 INTRODUCTION

The increasing popularity of scale out systems and the use of cheap commodity hardware with many CPU cores has given rise to elastic computing technologies that are inherently concurrent in nature, placing concurrency at the heart of fields such as Operating Systems, Distributed Computing, Database Systems and Computer Architecture. However, traditional systems programming languages such as C and C++ field programming constructs designed for an era prior to the evolution of multi-chip, multi-node hardware and software architectures. Additionally, developing and debugging concurrent software inherently challenging due to non-determinism. As shown by Stabros et. al. in [3] concurrency has become computationally expensive, the Lock Manager(detects and recovers from deadlocks) for an OLTP Database was responsible for 16.3% of the total CPU instructions for the database application. Strong programming language level concurrency constructs will not only enable strong run-time guarantees of concurrent programs but lower development overhead for concurrent programs. The Featherweight X10 paper by Lee et. al.[2] implements a core language subset of the X10 programming language, including the async and finish parallelism constructs. This core subset forms the basis for type system and static analyses for traceable proofs of correctness and solves the problem of context-sensitive may happen-in-parallel analysis for languages with async-finish parallelism. As stated in the paper[2] pairwise reachability problem for statements in a program is generally undecidable and NP Complete under certain conditions but the paper builds an approximation algorithm using a type system based static analyzer with low false positives. Such static analyzers can be used not just for static analysis of concurrent programs with async finish parallelism but in data race detectors as well. The programming language FX10 is a subset of X10. FX10 is defined in [2] includes rules for grammar, and a small-step operational semantics. We will formalize FX10 in Coq. The syntax will be implemented using inductive types and the semantics using a binary steps-to semantic. We will then prove the deadlock freedom of FX10 by proofing the progress theorem using the Coq proof assistant.

2 THE X10 LANGUAGE

The X10 language was designed specifically for parallel computing. Its core object-oriented features are very similar to those of Java or C#. It provides Classes, structs, interfaces and methods etc. For parallelism it provides the constructs *async* and *finish*. Async executes the the enclosing code-block in a newly spawned thread, while finish is used to wait and block until enclosing code-block completes execution in the current thread before executing the statements following the finish code block.

3 FEATHERWEIGHT X10

In [2], the authors describe a featherweight version of X10 called FX10. A featherweight version of a language makes it easy to formalize it in Coq and do proofs. The FX10 language borrows a subset of the X10 constructs such as while-loops, assignments, async, finish etc. In section 3 of [2], the authors also show that every FX10 program can be compiled to X10 by adding some boilerplate code.

4 SYNTAX

In FX10 the state of the program is defined as a 3-tuple of the *Program*, *Array* and an execution *Tree* as shown in (Figure 1).

$$
\begin{array}{lcl}
A \in Array & = & \mathbb{N} \to \mathbb{Z} \\
Tree: \quad T & ::= & T \triangleright T \quad | \quad T \parallel T \quad | \quad \langle s \rangle \quad | \quad \surd \\
State & = & Program \times Array \times Tree
\end{array}
$$

Figure 1: State in FX10: Shows how the state is constructed in FX10

```
Inductive program : Type :=
| p_cons (i:nat) (s : list statement)
.


Inductive tree : Type :=
| finished : tree
| sequential (t1 : tree) (t2 : tree) : tree
| parallel (t1 : tree) (t2 : tree) : tree
| exec (s : statement) : tree
.

Definition array := list Z.

Inductive state : Type :=
| cons_state (P : program) (A : array) (T : tree)
.
```

Figure 2: State encoded in CoQ

We define a global array as map from Natural numbers to integers with infinite length.

We simulate the mapping, meaning the infinite size array, with a finite data structure and helper methods. When an index that is out of bounds is read we return Z0 which is 0 of type integer and when an array write to an index that is out of bounds occurs we expand the array with Z0 until the value is inserted at the correct index. The write to array function is formalized in Coq as shown in Figure 8

A tree of the form T1 ▷ T2 executes T1 and T2 *sequentially*. While a tree of the form T1 || T2 executes T1 and T2 in *parallel*. The leaves in the AST are either $\surd$ or ⟨ s ⟩ where $\surd$ is finish and s is a statement.

More Syntax

In Figure 3 we introduce the full syntax for FX10. An FX10 program consists of a list of functions f_i, each with no arguments, return type void, and body s_i as well as the index of the main function in this list. The body of each function is a single statement. A statement is a sequence of labeled instructions ending with a skip statement. Each instruction is either skip, assignment, while loop, async, finish, or a method call. The right-hand side of an assignment is an expression that can be either an integer constant or an array lookup plus one. Additionally, skip instruction is defined as *skip_i* in our Coq program and *skip_s* is used as the terminal for statement. To avoid the chicken vs egg problem arising from Instruction and Statement referencing each other through mutual recursion we define both at the same time.

$$
\begin{array}{lcll}
Program: & p & ::= & void\ f_i()\ \{\ s_i\ \}, i \in 1..u \\
Statement: & s & ::= & skip^l \\
 & & | & i\ s \\
Instruction: & i & ::= & skip^l \\
 & & | & a[d] =^l e; \\
 & & | & while^l\ (a[d] \neq 0)\ s \\
 & & | & async^l\ s \\
 & & | & finish^l\ s \\
 & & | & f_i()^l \\
Expression: & e & ::= & c \\
 & & | & a[d] + 1
\end{array}
$$

Figure 3: The Grammar of Featherweight X10

```
Inductive expression : Type :=
| constant (c : Z) : expression
| a_inc (d : nat) : expression
.


Inductive statement : Type :=
| skip_s : statement
| is (i : instruction) (s : statement) : statement
with
instruction : Type :=
| skip_i : instruction
| a_write (d: nat) (e : expression) : instruction
| while (d : nat) (s: statement) : instruction
| async (s : statement) :instruction
| finish (s : statement) :instruction
| call (i:nat)
.


Inductive program : Type :=
| p_cons (i:nat) (s : list statement)
.
```

Figure 4: Syntax encoded in CoQ

5 SEMANTICS

In Coq as shown in as shown in Figure 6 we define *stepsto* rules as a binary relation, enabling us to perform induction on these rules. The evolution of programs in FX10 is through binary *stepsto* rules on every possible state in FX10. Figure 5 shows the steps to rule for trees that are made up of binary operators ▷ and ||. Figure 7 further defines the semantics when the tree is an execution of statement. Figure 8 shows how we write to an array location and evaluate expressions on arrays using the following rules:

$$A(c) = c$$

$$A(a[d] + 1) = A(c) + 1$$

where c is a constant that is read from a[d].

6 PROGRESS PROOF

Using the 14 steps to rules of the FX10 semantics the progress proof for FX10 shows that every possible state is either a

$$(p, A, \surd \triangleright T_2) \rightarrow (p, A, T_2) \quad (1)$$

$$\frac{(p, A, T_1) \rightarrow (p, A', T_1')}{(p, A, T_1 \triangleright T_2) \rightarrow (p, A', T_1' \triangleright T_2)} \quad (2)$$

$$(p, A, \surd \parallel T_2) \rightarrow (p, A, T_2) \quad (3)$$

$$(p, A, T_1 \parallel \surd) \rightarrow (p, A, T_1) \quad (4)$$

$$\frac{(p, A, T_1) \rightarrow (p, A', T_1')}{(p, A, T_1 \parallel T_2) \rightarrow (p, A', T_1' \parallel T_2)} \quad (5)$$

$$\frac{(p, A, T_2) \rightarrow (p, A', T_2')}{(p, A, T_1 \parallel T_2) \rightarrow (p, A', T_1 \parallel T_2')} \quad (6)$$

Figure 5: Stepsto Rules in FX10

```
Inductive stepsto : state -> state -> Prop :=
| step_1 p A T2 : stepsto
(cons_state p A (sequential finished T2)) (cons_state p A T2)
```

Figure 6: Coq Representation for Rule 1

$$(p, A, \langle skip^l \rangle) \rightarrow (p, A, \surd) \quad (7)$$

$$(p, A, \langle skip^l\ k \rangle) \rightarrow (p, A, \langle k \rangle) \quad (8)$$

$$(p, A, \langle a[d] =^l e; k \rangle) \rightarrow (p, A[c := A(e)], \langle k \rangle) \quad (9)$$

$$(p, A, \langle (while^l\ (a[d] \neq 0)\ s)\ k \rangle) \rightarrow (p, A, \langle k \rangle) \quad (\text{if } A(c) = 0) \quad (10)$$

$$(p, A, \langle (while^l\ (a[d] \neq 0)\ s)\ k \rangle) \rightarrow (p, A, \langle s\ .\ (while^l\ (a[d] \neq 0)\ s)\ k \rangle) \quad (\text{if } A(c) \neq 0) \quad (11)$$

$$(p, A, \langle (async^l\ s)\ k \rangle) \rightarrow (p, A, \langle s \rangle \parallel \langle k \rangle) \quad (12)$$

$$(p, A, \langle (finish^l\ s)\ k \rangle) \rightarrow (p, A, \langle s \rangle \triangleright \langle k \rangle) \quad (13)$$

$$(p, A, \langle f_i()^l\ k \rangle) \rightarrow (p, A, \langle s_i\ .\ k \rangle) \quad (\text{where } p(f_i) = s_i) \quad (14)$$

Figure 7: Semantics when the tree is of the form exec statement

```
Fixpoint get_from_array (a:array) (i:nat) :Z :=
match i,a with
|_,nil => 0
|O,cons x l => x
| S i', cons x l => get_from_array l i'
end.

Fixpoint eval_expr (a:array) (e:expression) : Z:=
match e with
| constant z => z
| a_inc d => (get_from_array a d)+1
end.

Fixpoint write_to_array (a:array) (i:nat) (val:Z):array:=
match i,a with
|O,nil => cons val nil
|S i',nil => cons  Z0 (write_to_array nil i' val)
|O,cons x l => cons val l
| S i', cons x l => cons x (write_to_array l i' val)
end.
```

Figure 8: Reading/Writing to Array and Defining array on expressions

value or it can step to a second state. FX10 defines a value as any state tuple where the tree's state is Finished(√). Deadlock freedom in FX10 follows from the progress proof(Figure 10) because getting stuck in a program without stepping to a new second state or finishing is not possible. It is important

```
Fixpoint get_function (i:nat) (p:program) : statement :=
match p with
| p_cons n s =>
(match i,s with
| _,nil => skip_s
| O, cons x l => x
| S i',cons x l => get_function i' (p_cons n l)
end)
end.


Theorem p_stays_same: forall (p1 p2 :program) (a1 a2 : array) (t1 t2 :tree),
  stepsto (cons_state p1 a1 t1) (cons_state p2 a2 t2) -> p1 = p2.
Proof.
  intros. inversion H; reflexivity.
Qed.
```

Figure 9: Getting a Function and proof for "P" Stays the same

(Deadlock freedom) For every state (p, A, T), either $T = \surd$ or there exists A', T' such that $(p, A, T) \rightarrow (p, A', T')$.

Figure 10: Every Possible state *s* is either a value or steps to *s'*

to note that *async* and *finish* constructs of FX10 do not guarantee data race avoidance despite guaranteeing deadlock freedom.

In Coq the progress proof is formally defined as:
Theorem progress: forall (s:state), value s exists s', stepsto s s'.
Proof.
Perfroming *induction* on the state *s* the proof first splits the state s into the program P, the array A, and the tree T. We then perform induction on the tree structure with *induction T*, which results in four cases: Finished(√), Sequential(▷), Parallel(||), and Exec(⟨ s ⟩).

Helper Functions and Theorems

Helper functions and theorems are defined to support the progress proof. Figure 9 shows two helper functions *get_function* and *p_stays_same*. *get_function* gets the body of a function at a specific index in a program. Note that in all the steps-to rules, the program variable remains the same. However, in our progress proof, the resulting Hypotheses have different values of P on each side. For such scenarios we use the P stays same theorem(as shown in Figure 9) to express our goal in terms of the same P.

Finished √

If a Tree in a State is in finished state, there is nothing left to prove, we apply the definition of value.

Sequential ▷

For sequential execution we destruct the induction-hypothesis for T1. Either T1 is a value or T1 is part of a state that can step to another state containing the tree T1'. The first case is proven by applying *step_1* and the second case by applying *step_2*. For the second case the *step_2* needs the program P to remain the same using the P stays the same proof(Figure 9) and then prove the premise of *step_2*.

***Parallel* ||**

The Parallel execution is proven very similarly to the Sequential case. After destructing the induction-hypothesis for T1 we use *step_3* if T1 is a value or *step_5* if T1 was part of a state that could step to a second state, and then prove the premise for *step_5*.

***Exec* ⟨ s ⟩**

We prove the Exec case by first using the destruct tactic in CoQ on the state which results in two sub-goals.

- *skip_s*, no instruction left
  - As all instructions end with a *skip_s*, if we are at a skip it implies we have executed all instructions. We simply apply *step_7*.
- *is*, instruction i left, use *destruct i.*
  - If the instruction is s skip instruction we apply *step_8*.
  - If the instruction is an array write we evaluate the expression and apply *step_9*.
  - If the instruction is a while loop:
    - The execution of the loop depends on the loop invariant, so we destruct on the loop invariant.
    - If the loop invariant is zero, we apply *step_10*.
    - If the loop invariant is not zero, we apply *step_11*.
  - If the instruction is an async we apply *step_12*.
  - If the instruction is a finish we apply *step_13*.
  - If the instruction is a function call we apply *step_14*.

7 CONCLUSION

We presented the syntax and semantics for FX10 and formalized its syntax and the steps-to semantics in Coq. Leveraging Coq's proof management constructs we show the correctness of FX10 and prove progress theorem for all possible states of a program and hence prove the absence of deadlocks. For future work in order to perform may-happen-in-parallel analysis for FX10 we can take up formalizing and proving the correctness of the type system as presented in [2].

REFERENCES

[1] Divyanshu Upreti Florian Peter Donhauser, James Rohan Gangavarapu. 2019. Featherweight X10 Project Repository. https://github.gatech.edu/fdonhauser3/CS6390FX10

[2] Jonathan K Lee and Jens Palsberg. 2010. Featherweight X10: a core calculus for async-finish parallelism. In *ACM Sigplan Notices*, Vol. 45. ACM, 25–36.

[3] Samuel Madden Stavros Harizopoulos, Daniel J. Abadi and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. MIT.