Abstract

In this project we will formalize and prove Theorem 1 of Featherweight X10\(^1\) for an abstract expression language. We will also formalize a notion of sequential programs and show that such a property is preserved by small-step semantics.

1 Language

(Text adapted from the FX10 paper.)

The semantics of FX10 uses the binary operator || in the semantics of async, it uses the binary ▷ operator in the semantics of finish, and it uses the constant √ to model a completed computation. A state in the semantics consists of a tree \(T\) that describes the code executing. The internal nodes of \(T\) are either || or ▷, while the leaves are √ or ⟨\(s\)⟩, where \(s\) is a statement.

As an example of how the semantics works, we will now informally discuss an execution of a program. The execution begins with a finish statement.

\[
\langle \text{finish} \{\text{async} \{s_2\}; s_3\}; s_1\rangle \Rightarrow \\
\langle \text{async} \{s_2\}; s_3 \triangleright \langle s_1\rangle \Rightarrow \\
\langle s_2 \rangle || \langle s_3 \rangle \triangleright \langle s_1\rangle \Rightarrow
\]

The first step illustrates the semantics of finish and introduces ▷ to signal that the left-hand side of ▷ must complete execution before the right-hand can proceed. The second step illustrates the semantics of async and introduces || to signal that \(e_2\) and \(e_3\) should proceed in parallel. The two sides of || can execute in parallel, which we model with an interleaving semantics. When one of the sides completes execution, it will reach the state √. For example if \(\langle s_3 \rangle \Rightarrow √\), then the semantics can do \(\langle s_2 \rangle || \langle s_3 \rangle \Rightarrow \langle s_2 \rangle || √ \Rightarrow \langle s_2 \rangle\). When also \(s_2\) completes execution, the semantics can finally proceed with the right-hand side of ▷, that is \(\langle s_1\rangle\).

A statement is a sequence of instructions. Each instruction is either skip, evaluating an expression \(e\), async, or finish.

\[
s ::= \text{skip} \mid e; s \mid \text{async} \{s\}; s \mid \text{finish} \{s\}; s
\]

An async statement \(\text{async} \{s_1\}; s_2\) runs \(s_1\) in parallel with the continuation of the async statement \(s_2\). The async statement is a lightweight notation for spawning threads, while a finish statement \(\text{finish} \{s_1\}; s_2\) waits for termination of all async bodies started while executing \(s_1\) before executing the continuation \(s_2\).

A tree $T_1 \triangleright T_2$ is convenient for giving the semantics of finish: $T_1$ must complete execution before we move on to executing $T_2$. A tree $T_1 \| T_2$ represents a parallel execution of $T_1$ and $T_2$ that interleaves the execution of subtrees, except when disallowed by $\triangleright$. A tree $\langle s \rangle$ represents statement $s$ running. A tree $\sqrt{\ }$ has completed execution.

2 Small-step semantics

<table>
<thead>
<tr>
<th>Rules for statements $s \Rightarrow T$</th>
<th>Rules for trees $T \Rightarrow T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e \Rightarrow e'$</td>
<td>$\sqrt{\triangleright} T \Rightarrow T$</td>
</tr>
<tr>
<td>$e; s \Rightarrow \langle e'; s \rangle$</td>
<td>$T_1 \Rightarrow T_1'$</td>
</tr>
<tr>
<td></td>
<td>$T_1 \triangleright T_2 \Rightarrow T_1' \triangleright T_2$</td>
</tr>
<tr>
<td>$\text{value}(e)$</td>
<td>$\sqrt{|} T \Rightarrow T$</td>
</tr>
<tr>
<td>$e; s \Rightarrow \langle s \rangle$</td>
<td>$T | \sqrt{\ } \Rightarrow T$</td>
</tr>
<tr>
<td>skip $\Rightarrow \sqrt{\ }$</td>
<td>$T | \sqrt{\quad} \Rightarrow T$</td>
</tr>
<tr>
<td>$\text{async}{s_1}; s_2 \Rightarrow \langle s_1 \rangle | \langle s_2 \rangle$</td>
<td>$T_1 \Rightarrow T_1'$</td>
</tr>
<tr>
<td>$\text{finish}{s_1}; s_2 \Rightarrow \langle s_1 \rangle \triangleright \langle s_2 \rangle$</td>
<td>$T_1 | T_2 \Rightarrow T_1' | T_2$</td>
</tr>
<tr>
<td>$s \Rightarrow T$</td>
<td>$T_2 \Rightarrow T_2'$</td>
</tr>
<tr>
<td>$\langle s \rangle \Rightarrow T$</td>
<td>$T_1 | T_2 \Rightarrow T_1' | T_2'$</td>
</tr>
</tbody>
</table>


3 Exercises

The homework shall be submitted via Blackboard as a single Coq file, named FX10.v.

**Exercise 1 (60%)**: Formalize the small-step semantics and show that it enjoys strong progress. You will need to assume that the abstract expression language $e$ enjoys strong progress.

**Theorem (Strong progress)**. For every state $T$, either $T = \sqrt{\_}$ or there exists $T'$ such that $T \Rightarrow T'$.

**Exercise 2 (10%)**: Prove that the FX10 language you have just defined can be instantiated with `Smallstep.tm` and prove that it enjoys strong progress. (The proof should be a simple application of the theorem of Exercise 1.)

**Exercise 3 (30%)**: We want to be able to identify statically sequential trees. Write a type system $\vdash T$ that rules out statements with `async` and trees with the parallel composition `||`, and show that such a type system enjoys type preservation. Finally, show that the type system is inhabited, that is, there exists at least one tree that is well-typed.
Require Import Smallstep.

Section FX10.

(* Abstract expressions are parameters of our theory. *)
Variable exp : Set.
Variable e_step : exp -> exp -> Prop.
Variable value : exp -> Prop.
Variable exp_progress : forall x, value x / exists y, e_step x y.

(* Define our language *)
Inductive stmt : Set := (* TODO *)
Inductive tree : Set := (* TODO *)

(* Define the small-steps semantics *)
Inductive s_step:
(* TODO: small-step relation for statements *)
Inductive t_step:
(* TODO: small-step relation for trees *)
(* Exercise 1: *)
Theorem t_strong_progress:
(* TODO: Prove strong progress for [t_step] *)
End FX10.

Inductive s_seq: (* TODO: type system for statements *)
Inductive t_seq: (* TODO: type system for trees *)
(* Exercise 2: *)
Lemma subject_reduction:
(* TODO: Prove subject reduction *)
(* Exercise 3: *)
Lemma tm_strong_progress:
(* TODO: Prove strong progress for [t_step] parameterized with [tm] *)