

Project 2 – F^{*}

TU Wien

Group 5

Our group consists of the following members:

Tobias Eidelpes
01527193

Exercise 2.1: Small-step semantics

Notes on semantics. Not mandatory.

Exercise 2.2: Termination

Explanation of the used measure The measure consists of a list of size three. The entries are:

1. The amount of gas in the execution state.
2. The size of the call stack.
3. If the call stack is constructed with **Exec**, the third element is the natural number 1 and if the call stack is constructed with **Ter**, the third element is the natural number 0.

For the amount of gas, there are three different possibilities:

1. The call stack is a plain call stack.
2. If the call stack is in a halting state, gas specifies the amount of remaining gas.
3. If there is an exception state, the amount of gas is set to 0.

Additionally, there are three match clauses in the **step** function. The first case matches a termination state with an empty call stack. If the state before that was an exception, the first elements are equal but the size of the call stack before the application of the **step** function was greater. This means that the measure decreases. If there was no exception but of type **Exec** with a positive amount of gas, the measure also decreases.

In the second match clause, a termination state with a non-empty call stack is handled. The amount of gas decreases if there is a termination state of **ExcState**, because it is set to 0. If the termination state is **HaltState**, the amount of gas also decreases.

The third match clause handles execution of a plain call stack. If a **HaltState** is reached, the amount of gas reduces by 1. Again, if there occurs an exception, looking at the third element of the list gives a value change from 1 to 0.

Exercise 2.3: Final states

Paper proofs To prove `nsteps_stop`, induction over the amount of executed steps n is used. The base case starts with $n = 0$. This case is trivial since `nsteps 0 te cs == cs`. The induction hypothesis is `nsteps (n-1) te cs == cs` and it is assumed that $n > 0$. The induction then goes as follows:

```
nsteps n te cs = nsteps (n-1) te (step_simp te cs)
                = nsteps (n-1) te cs
                = cs
```

It is allowed to reduce `step_simp te cs` to `cs`, because `cs` is already final.

Now it is possible to prove the lemma `progress`: `cs == step_simp te cs → isFinal cs`. Since `step_simp` has the same value as the `step` function, when the outcome is `Next`, the call stack is different from `cs`. This means that the left side of the lemma does not hold and therefore the right side does not have to hold to satisfy the implication. If the outcome is `Stop`, `cs` can only equal `Ter ts []` and therefore the left side is satisfied. When the left side is satisfied, `cs` is a final state and therefore it follows that `isFinal cs` is also true.

Explanation on used equalities The lemma uses propositional equality, because `callstack` has the keyword `noeq`, which means that the type does not satisfy decidable equality. The usual equality only works with types that satisfy decidable equality and therefore propositional equality is needed.

Exercise 2.4: Uniqueness of call stacks

Paper proofs Note: The previously defined measure is written as `|cs|`, where `cs` is a call stack.

To prove `order_decreases` it needs to be shown that from `nsteps n te cs == cs'` and $n > 0$ follows `|cs'| << |cs|` for all $n \in \mathbb{N}$. The proof is by induction on n . For $n = 1$, `nsteps 1 te cs` is equal to `step_simp te cs`. Similarly to Exercise 2.2 it follows that the hypothesis holds, because `cs` is not a final state. After that, the induction step is `nsteps (n+1) te cs == cs'`, which can be rewritten as `nsteps n te (step_simp te cs) == cs'` (definition of `nsteps`). With the reasoning from the base case where $n = 1$, it follows that `|step_simp te cs| << |cs|`. Now there are two cases: either the call stack `|step_simp te cs|` is final or not. For the first case, the lemma `nsteps_stop` can be used to derive `step_simp te cs == cs'` from `nsteps n te (step_simp te cs) == cs'`, where `|step_simp te cs| << |cs|` holds. For the second case, we simply take the induction hypothesis to conclude that the induction holds.

Taking the previous proof, for `callstacks_unique` it is evident that `|cs'| << |cs|` holds. It follows that `cs' ≠ cs` by proof of contradiction: Assuming that `cs' == cs` and `|cs'| << |cs|` is true, it would require that the top constant in both lists `LexTop` satisfies `LexTop << LexTop`. Since this is impossible, it contradicts the assumption and therefore it follows that `cs' ≠ cs`.

Exercise 2.5: Exception Propagation

Paper proof Proof by induction:

```
length ps = 0
```

```

nsteps (2 * (length [])) te (Ter ExcState []) == (Ter ExcState [])
nsteps 0 te (Ter ExcState []) == (Ter ExcState [])
(Ter ExcState []) == (Ter ExcState [])

```

```

length ps to (length ps) + 1

```

```

nsteps (2 * (length ps)) te (Ter ExcState ps) == (Ter ExcState [])
nsteps ((2 * (length ps)) - 1) te (step simp (Ter ExcState ps)) == (Ter ExcState [])
nsteps ((2 * (length ps)) - 1) te (Exec (apply returneffects ts s)::ps') == (Ter ExcState ps)
nsteps ((2 * (length ps)) - 1) te (Exec s'::ps') == (Ter ExcState ps)
nsteps ((2 * (length ps)) - 2) te (step simp (Exec s'::ps')) == (Ter ExcState ps)
nsteps ((2 * (length ps)) - 2) te (Ter (ExcState ps')) == (Ter ExcState ps)
nsteps (2 * ((length ps) - 1)) te (Ter (ExcState ps')) == (Ter ExcState ps)
nsteps (2 * (length ps')) te (Ter (ExcState ps')) == (Ter ExcState ps)

```