Exam
Essentials of Programming Languages, 2020 SS

Prof. Dr. Peter Thiemann
Hannes Saffrich

August 26, 2020

Submission Deadline: October 15, 2020 (via email[1])

In the chapter *Lambda*, we defined the syntax, semantics, and type system of the Simply Typed Lambda Calculus (STLC), extended with natural numbers and μ-recursion.

In the chapter *Properties*, we then proved the *progress* and *preservation* lemmas, which together correspond to *type soundness*, i.e. that the type system correctly describes the semantics. Type soundness is essential for programmers to rely on the type system: if the type system decides that a program has type $\mathbb{N}$, then type soundness guarantees that running the program will never crash, and either return a value of type $\mathbb{N}$ or run forever. In our formalization, a program crashes if after some number of evaluation steps, the program is neither a value nor can be evaluated any further, e.g. a program which tries to use a number as a function.

This exam consists of two exercises, in which you are going to extend the beforementioned formalization in different ways. In those exercises you must make various proofs based on Agda code which we provide. For the more complicated proofs, which require lemmas, we not only state the theorem you need to prove, but also the lemmas we proved ourself to solve the exercise. You do not have to prove or use those lemmas, but it is strongly recommended. In case you only partially finish an exercise, proving the lemmas still counts as progress. If you design your own lemmas, then they only count if they actually work towards the main proofs required by the exercise.

Exercise 1: Big-Step Semantics

The semantics, which we have defined in the lecture, is a so-called *small-step semantics*: we think of the evaluation of a program as a potentially infinite sequence of small finite steps $\_ \to \_$.

$$(1 + (2 + 3)) \to (1 + 5) \to 6$$

[1] saffrich@informatik.uni-freiburg.de

An alternative way to describe evaluation is by a so-called *big-step semantics*, where the evaluation of a program is defined as a relation $\_ \Downarrow \_$ whichs connects the program directly to its final value:

$$(1 + (2 + 3)) \Downarrow 6$$

In this exercise, we extend the formalization from the lecture by

- defining a big-step semantics for the language;
- proving that the type system is sound for the big-step semantics;
- proving that the big-step and the small-step semantics are equivalent.

As a starting point we have prepared a set of `.agda`-files for you, containing the formalization from the lecture, a definition of the big-step semantics, and skeletons and hints for the lemmas you need to prove.[2]

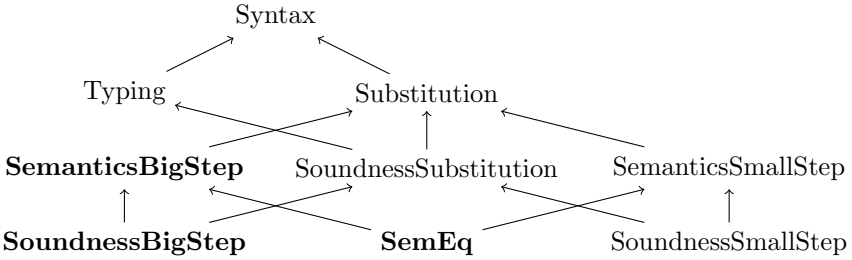

Figure 1: Dependencies of the `.agda`-files in `Exam/Exercise1/`.

The dependencies between those `.agda`-files are shown in Figure 1. The files in regular font contain only code from the lecture/book:

- *Syntax.* Defines the syntax of `Term`s.
- *Substitution.* Defines the substitution function `_[_:=_]`.
- *SemanticsSmallStep.* Defines the small-step semantics relation $\_ \longrightarrow \_$, its reflexive, transitive closure $\_ \longrightarrow\!\!\!\!\!\rightarrow \_$, and what it means for a `Term` to be a `Value`.
- *Typing.* Defines the syntax of `Type`s and `Context`s, the typing relation $\_ \vdash \_ : \_$, and `Context`-membership $\_ \ni \_ : \_$.
- *SoundnessSubstitution.* Proves that the typing relation is preserved under substitution, i.e. the `subst-preserve` lemma.

[2] It's probably a good idea, to take some time to familiarize yourself with the codebase, in particular to identify the definitions from the lecture, and to understand the new definitions, instead of directly trying to prove the lemmas.

– *SoundnessSmallStep*. Proves type soundness by proving the `preservation` and `progress` lemmas. In the `preservation` proof, the `subst-preserve` lemma from *SoundnessSubstitution* is used.

The files in **bold** font contain new definitions and placeholders for your solutions:

– ***SemanticsBigStep***. Defines the big-step semantics, and shows examples of how it works.

– ***SoundnessBigStep***. In this file you should prove the soundness of the big-step semantics. (Further instructions included.)

– ***SemEq***. In this file you should prove that big-step and small-step semantics are equivalent, and then use this equivalence to derive a simpler soundness proof for the big step semantics. (Further instructions included.)

Your task:

– In `Exam/Exercise1/SoundnessBigStep.agda`, prove the `soundness` theorem.

– In `Exam/Exercise1/SemEq.agda`, prove the `to→⇛`, `to⇓`, and `soundness'` theorems.

– If you use the lemmas, which we propose as `postulate`s, you also need to prove them.

Exercise 2: Mutable References

In this exercise we are going to extend the language from the lecture with a feature called *mutable references* and prove type soundness for the extended language.

Mutable references are a central feature of any imperative programming language. In C, they are expressed as pointers into heap allocated memory; in object oriented languages, they appear as object allocation and getting/setting the value of an object's field. Consider for example the following Java code:

```
class Cell {
  int value;
}

Cell c = new Cell(42);    // allocation
print(c.value);           // read, prints 42
c.value = 23              // write
print(c.value);           // read, prints 23
```

In our language extension, we focus on the essence behind mutation: allocation of memory, reading from a memory address, and writing to a memory

address. In particular, we do not model classes and objects, so allocated memory always stores a single value, instead of the values of multiple fields.

Hence, the previous Java example corresponds to the following code in our language:

```
let r = new 42 in
let x = (! r) in
let _ = (r := 23) in
let y = (! r) in
y
```

Running this program behaves as followed:

- In line 1, the term `new 42` creates a new memory cell containing the value `42`, and returns the memory cell's location. The returned location is simply a natural number representing a memory address (also called a reference or pointer value).
- In line 2, the term `! r` retrieves the value currently stored at location `r`, so `x` is going to be `42`.
- In line 3, the term `r := 23` replaces the value stored at location `r` with the new value `23`, and returns the unit value, which we ignore.[3]
- In line 4, we again retrieve the value stored at location `r`, which now yields the updated value, so `y` is going to be `23`.

An important detail is that the content of the memory cells is not part of the program, but instead part of a separate data structure which we call a store. Hence, the semantics does not describe how one term reduces to another, but how one term with a certain store reduces to another term with an updated store, which we write as $\langle \_, \_ \rangle \to \langle \_, \_ \rangle$. We define a store as a list of values, where the value at index ℓ describes the value of the memory cell at location ℓ.

For the example above this means the following:

- In line 1, the term `new 42` is evaluated in the empty store $[]$, and yields an updated store containing value 42 at a new location:

  $$\langle [], \texttt{new 42} \rangle \to \langle [42], \texttt{loc 0} \rangle.$$

  In this case, we store the value 42 at location 0, since the initial store was empty.
- In line 2, we then evaluate `! r` in the new store $[42]$, where `r` has been replaced by `loc 0`:

  $$\langle [42], \texttt{!(loc 0)} \rangle \to \langle [42], 42 \rangle.$$

[3] Returning the unit value has the same purpose as defining functions of type `void` in Java or C. It signals that the function returns nothing of interest, but instead has interesting side-effects, like printing on the terminal or changing the state of memory.

Retrieving the value at location 0, amounts to looking up the list element at index 0 of the store. The store itself remains unchanged.

– In line 3, we then evaluate `r := 23`, where `r` has been replaced by `loc 0`:

$$\langle [42], \texttt{loc 0 := 23} \rangle \to \langle [23], \texttt{tt} \rangle.$$

Here we want to update the value at location 0, so we go from the initial store [42] to the new store [23]. The term itself reduces to the unit value `tt`.

– Line 4 works analogously to line 2, but starts from a different initial store.

As for the previous exercise, we have prepared a set of `.agda`-files for you, containing the formalization from the lecture extended with mutable references, and skeletons and hints for the lemmas you need to prove.

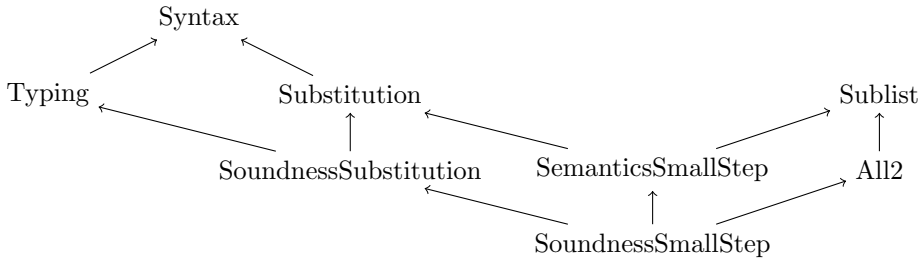

Figure 2: Dependencies of the `.agda`-files in `Exercise3`.

The dependencies between those `.agda`-files are shown in Figure 2. The new files are

– *Sublist.* Contains functions and lemmas about lists, which we need in the semantics definition to describe how the store is modified. The file suggests 4 lemmas and 2 bonus exercises for you to prove.

– *All2.* Contains the definition and lemmas about the $\texttt{All}_2$ type, which is similar to the `All` type from the lecture, but for binary relations. The file suggests 3 lemmas for you to prove.

The other files serve the same purpose as in Exercise 1, but are extended with the constructs of mutable references and plenty of comments. The following of those files contain lemmas for you to prove:

– *SoundnessSubstitution.* Contains 2 lemmas to prove.

– *SoundnessSmallStep.* Contains 3 lemmas to prove, and the actual preservation theorem.

Your task:

- In `Exam/Exercise2/SoundnessSmallStep.agda`, prove the `preserve` theorem.
- If you use the lemmas, which we propose as `postulate`s, you also need to prove them.