Homework #2

CS231

Due by the end of the day on January 30. You should submit two files:
`hw2.ml` and `hw2.pdf`.

Remember that you are encouraged to work in pairs on homework assignments. See the course syllabus for details. Also remember the course's academic integrity policy. In particular, you must credit other people and other resources that you consulted. Again, see the syllabus for details.

1. In the `hw2.ml` file I've defined the type `t` of terms for the language of booleans and numbers, just as we had in the previous homework. I've also defined the type `typ` of types for this language. Implement the function `typecheck`, which typechecks a given term. That is, `typecheck t` should produce the type `T` if and only if $\texttt{t} : \texttt{T}$ according to the typechecking rules that we saw in class (see the Homework #2 Cheat Sheet). Your function should raise the `TypeError` exception if no type can be derived for the given term, according to our rules.

2. Consider the small-step operational semantics for booleans and integers in Section 1.2 of the cheat sheet. Provide a grammar for a new metavariable `s` that characterizes exactly the stuck expressions relative to this semantics — the set of terms that are not values but cannot step. You can introduce other metavariables as needed.

3. Like C, OCaml includes terms of the form $\texttt{t}_1$ `&&` $\texttt{t}_2$, which represent short-circuited "and" for booleans: the second boolean expression is evaluated only if necessary. In this problem we add terms of this form to our language of booleans and numbers.

   (a) Add rules to the small-step operational semantics for our language (see the cheat sheet) to support the new kind of term. You should give a *direct* semantics for this new term rather than simply "desugaring" it to some other kind of term (we'll do that later). By a *direct* semantics, we mean that all terms of the form $\texttt{t}_1$ `&&` $\texttt{t}_2$ should either step to another term of that same form, to a subterm of the original term, or to a value. Give a name to each new rule.

   (b) Add rules to the type system to support the new operator. Give a name to each new rule.

   (c) Provide only the cases specific to the new $\texttt{t}_1$ `&&` $\texttt{t}_2$ term for the proof of the Progress theorem:

   Theorem: If $\texttt{t} : \texttt{T}$, then either `t` is a value or there exists some term $\texttt{t}'$ such that $\texttt{t} \longrightarrow \texttt{t}'$.

   Assume that the proof is performed by induction on the derivation of $\texttt{t} : \texttt{T}$. If your proof uses a Canonical Forms lemma, state the lemma clearly before your proof (but you need not prove the lemma).

   (d) Provide only the cases specific to the new $\texttt{t}_1$ `&&` $\texttt{t}_2$ term for the proof of the Preservation theorem:

   Theorem: If $\texttt{t} : \texttt{T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\texttt{t}' : \texttt{T}$.

   Assume that the proof is performed by induction on the derivation of $\texttt{t} : \texttt{T}$.

4. (a) It turns out that terms of the form $\texttt{t}_1$ `&&` $\texttt{t}_2$ can be treated as "syntactic sugar," or shorthands for other terms in the language of booleans and numbers. Demonstrate this by providing a new operational semantics for these terms, which consists of a single rule with no premises.

(b) Consider an eager version of `&&`, in which both operands are evaluated (in order from left to right) before producing the overall value of the term. Is this version still a syntactic sugar? If so, provide the semantics. If not, just say so.

5. Consider each of the following changes to the language of booleans and integers in Section 1 of the cheat sheet. For each change, say whether it invalidates Progress, Preservation, both, or neither. Also provide a counterexample to each invalidated theorem.

   (a) Remove the rule E-IFFALSE.

   (b) Add the following axiom to the type system:

$$\frac{}{\texttt{0:Bool}}$$

   (c) Add the following axiom to the operational semantics:

$$\frac{}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow t_2}$$

   (d) Add the following rules:

$$\frac{}{\texttt{false + false} \longrightarrow \texttt{false}}$$

$$\frac{}{\texttt{true + true} \longrightarrow \texttt{true}}$$

$$\frac{t_1\texttt{:Bool} \qquad t_2\texttt{:Bool}}{t_1 \texttt{ + } t_2\texttt{:Int}}$$

   (e) Add the following rules:

$$\frac{}{\texttt{if 0 then } t_2 \texttt{ else } t_3 \longrightarrow t_2}$$

$$\frac{t_1\texttt{:Int} \qquad t_2\texttt{:T} \qquad t_3\texttt{:T}}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3\texttt{:T}}$$

6. Consider again Section 1 of the cheat sheet.

   (a) Consider this "reverse progress" theorem:
   Theorem: If $\texttt{t}'\texttt{:T}$, then there exists some term `t` such that $\texttt{t} \longrightarrow \texttt{t}'$.
   Is this a true theorem? If so, prove it. If not, give a counterexample.

   (b) Consider this "reverse preservation" theorem:
   Theorem: If $\texttt{t}'\texttt{:T}$ and $\texttt{t} \longrightarrow \texttt{t}'$, then $\texttt{t:T}$.
   Is this a true theorem? If so, prove it. If not, give a counterexample.

7. Recall the big-step semantics judgment of the form `t` $\Downarrow$ `v`. Unfortunately there's no way to express the type soundness theorem — well-typed terms are not eventually stuck — in terms of this semantics. The problem is that this semantics does not distinguish between eventually stuck terms and non-terminating terms — for both kinds of terms `t`, there is no `v` such that `t` $\Downarrow$ `v`.

   In this problem you'll explore a way to modify the big-step semantics to solve this problem. Specifically, you'll define a new big-step semantics relation of the form `t` $\Downarrow_k$ `v`, where k is a nonnegative integer.

   (a) First provide modified versions of each of the big-step rules for booleans and integers on the cheat sheet to use the new big-step semantics relation. Your rules for `t` $\Downarrow_k$ `v` should be identical to the original rules, except that `t` $\Downarrow_k$ `v` should only be derivable when the corresponding derivation tree has height at most k (i.e., the number of nodes from the root to any leaf, inclusive, is at most k). For example, `1 + (2 + 3)` $\Downarrow_3$ `6` and `1 + (2 + 3)` $\Downarrow_{14}$ `6` should be derivable, but `1 + (2 + 3)` $\Downarrow_2$ `6` should not be derivable, since the derivation tree has height 3.

   As an example, here's the modification to the first big-step rule from the cheat sheet:

$$\frac{k \geq 1}{\texttt{v} \Downarrow_k \texttt{v}}$$

   (b) With these new rules, we still can't distinguish eventually stuck terms from non-terminating terms.[1] Now you will add new rules to your semantics to do so.

   First we'll add a new term to the grammar of the language, called `timeout`. Next we will add a rule to explicitly handle the situation when we run out of our height budget:

$$\frac{}{\texttt{t} \Downarrow_0 \texttt{timeout}}$$

   To finish the rules, add new rules that propagate the `timeout` whenever it occurs, so that programs that time out anywhere during their execution evaluate to `timeout`. For example, it should be possible using your rules to derive `1 + (2 + 3)` $\Downarrow_2$ `timeout`.

   Important Note! By convention we will assume that the premises in a big step rule "execute" in order from left to right. This is important because it affects which terms time out. For example, though `1 + (2 + 3)` $\Downarrow_2$ `timeout` should be derivable, `false + (2 + 3)` $\Downarrow_2$ `timeout` should *not* be derivable, since the program gets stuck while executing the first premise of B-PLUS (since `false` is not a number) and so never executes the expression that would time out. On the other hand, `false + (2 + 3)` $\Downarrow_1$ `timeout` *is* derivable, since the execution of `false` itself times out.

   (c) Finally, use your new relation to properly state the type soundness theorem — that well-typed terms are not eventually stuck. (No need to prove the theorem; just state it.)

[1] The language of booleans and integers has no non-terminating terms, but the approach we are pursuing is a general way of solving the problem, for any language.