Constructive Logic (15-317), Spring 2022
Assignment 6: Classical Logic and Type Checking

Instructor: Klaas Pruiksma
TAs: Runming Li, Onyekachi Onyeador, Viraj Puri, Xiao Yu

Due: Submit to Gradescope by Thursday March 3, 11:59 pm

This assignment contains both coding and written portions. Written PDFs, Dcheck code files, and SML code files will both go to Gradescope. The written is required to be typeset. We recommend using LaTeX but other suitable tools are also acceptable.

- `hw6.deriv` (your coding solutions to classical logic)
- `handin.zip` (your coding solutions to type checking)
- `hw6.pdf` (your written solutions)

The coding portion will use the experimental Dcheck derivation checker. You can find documentation and examples on the Software page at the course web site (`https://www.andrew.cmu.edu/user/kpruiksm/15317s22/dcheck.pdf`).

1 DeMorgan's Revenge

Provide derivations of the following Classical Logic judgements using Dcheck syntax in system `CL`.

Task 1 (6 points)**.** Define a derivation named `task1` that derives:

$$\neg(A \wedge B) \supset (\neg A \vee \neg B) \ \mathsf{true}$$

Task 2 (6 points)**.** Define a derivation named `task2` that derives:

$$(A \supset B) \supset (\neg A \vee B) \ \mathsf{true}$$

Note that neither of these are constructively true in general.

2 Classical or Constructive

We observe that anything true in constructive logic is also true in classical logic, but not vice versa. Both of the following judgements are classically true, but are they constructively true as well? State in `hw6.pdf` whether or not they are constructively true. Then provide derivations in classical logic using Dcheck syntax in `hw6.deriv`.

Task 3 (6 points)**.** Define a derivation named `task3` in system `CL` that derives:

$$(A \supset \neg A) \supset \neg A \ \mathsf{true}$$

Task 4 (6 points)**.** Define a derivation named `task4` in system `CL` that derives:

$$((A \supset B) \supset A) \supset A \ \mathsf{true}$$

3 Classical Quantifiers

We can extend classical logic with universal and existential quantifiers by adding the following truth and falsity rules:

$$\dfrac{\begin{array}{c}[a:\tau]\\ \vdots \\ A(a)\ \mathsf{true}\end{array}}{\forall x{:}\tau.\ A(x)\ \mathsf{true}}\ \forall T^a \qquad\qquad \dfrac{t:\tau \quad A(t)\ \mathsf{false}}{\forall x{:}\tau.\ A(x)\ \mathsf{false}}\ \forall F$$

$$\dfrac{t:\tau \quad A(t)\ \mathsf{true}}{\exists x{:}\tau.\ A(x)\ \mathsf{true}}\ \exists T \qquad\qquad \dfrac{\begin{array}{c}[a:\tau]\\ \vdots \\ A(a)\ \mathsf{false}\end{array}}{\exists x{:}\tau.\ A(x)\ \mathsf{false}}\ \exists F^a$$

Note the duality between the $\forall$ and $\exists$.

Task 5 (10 pts)**.** Using these rules, show that the usual *elimination* rules for the universal and the existential quantifier are derivable. For reference, those rules are:

$$\dfrac{t : \tau \quad \forall x{:}\tau.\, C(x)\ \mathsf{true}}{C(t)\ \mathsf{true}}\ \forall\mathsf{E} \qquad\qquad \dfrac{\exists x{:}\tau.\, A(x)\ \mathsf{true} \quad \begin{array}{c}[a : \tau] \quad [A(a)\ \mathsf{true}]_u\\ \vdots\\ C\ \mathsf{true}\end{array}}{C\ \mathsf{true}}\ \exists\mathsf{E}^{a,u}$$

4 Type Checking

A Dcheck Extension

In Fall 2022 semester, you become a Clogic TA. You notice that the Dcheck autograder the course uses does not support a proof term system, but there is no reason why it can't! In order to save yourself hours of tedious manual grading, you decide to implement a proof term checker yourself.

Task 6 (66 points)**.** Your goal is to implement the function `check : exp -> prop -> bool` that returns `true` if the proof term (represented as type `exp`) is a correct proof of the proposition (represented as type `prop`), and `false` otherwise. This function should be implemented in `dist/checker/SimpleLC_Chec`
Proof terms are in the form of simply typed lambda calculus (STLC), the specification can be found below.

<table>
<tr><th></th><th></th><th>abstract syntax</th><th>concrete syntax</th><th>ML datatype</th><th>description</th></tr>
<tr><td><code>typ</code> τ</td><td>$::=$</td><td>A</td><td><code>A</code></td><td><code>AtomTy AA</code></td><td>base type</td></tr>
<tr><td></td><td>$\mid$</td><td><code>unit</code></td><td><code>unit</code></td><td><code>UnitTy</code></td><td>unit type</td></tr>
<tr><td></td><td>$\mid$</td><td><code>void</code></td><td><code>void</code></td><td><code>VoidTy</code></td><td>empty type</td></tr>
<tr><td></td><td>$\mid$</td><td>$\tau_1 \times \tau_2$</td><td><code>t1 * t2</code></td><td><code>Times (t1, t2)</code></td><td>product type</td></tr>
<tr><td></td><td>$\mid$</td><td>$\tau_1 + \tau_2$</td><td><code>t1 + t2</code></td><td><code>Plus (t1, t2)</code></td><td>sum type</td></tr>
<tr><td></td><td>$\mid$</td><td>$\tau_1 \to \tau_2$</td><td><code>t1 -> t2</code></td><td><code>Arrows (t1, t2)</code></td><td>function type</td></tr>
<tr><td><code>exp</code> e</td><td>$::=$</td><td>x</td><td><code>x</code></td><td><code>Variable "x"</code></td><td>variable</td></tr>
<tr><td></td><td>$\mid$</td><td>$\langle\rangle$</td><td><code>()</code></td><td><code>Unit</code></td><td>unit</td></tr>
<tr><td></td><td>$\mid$</td><td>$\langle e_1, e_2\rangle$</td><td><code>(e1, e2)</code></td><td><code>Tuple (e1, e2)</code></td><td>tuple</td></tr>
<tr><td></td><td>$\mid$</td><td>$\mathtt{fst}(e)$</td><td><code>fst e</code></td><td><code>First e</code></td><td>first tuple element</td></tr>
<tr><td></td><td>$\mid$</td><td>$\mathtt{snd}(e)$</td><td><code>snd e</code></td><td><code>Second e</code></td><td>second tuple element</td></tr>
<tr><td></td><td>$\mid$</td><td>$\mathtt{inl}_{\tau_1+\tau_2}(e)$</td><td><code>inl e into t1 + t2</code></td><td><code>Inl (e, (t1, t2))</code></td><td>left injection</td></tr>
<tr><td></td><td>$\mid$</td><td>$\mathtt{inr}_{\tau_1+\tau_2}(e)$</td><td><code>inr e into t1 + t2</code></td><td><code>Inr (e, (t1, t2))</code></td><td>right injection</td></tr>
<tr><td></td><td>$\mid$</td><td>$\mathtt{case}(e; x_1.e_1, x_2.e_2)$</td><td><code>case e of</code>
<code> inl x1 => e1</code>
<code>| inr x2 => e2</code></td><td><code>Case (e,</code>
<code> ("x1", e1),</code>
<code> ("x2", e2))</code></td><td>case expression</td></tr>
<tr><td></td><td>$\mid$</td><td>$\lambda(x : \tau).e$</td><td><code>fn (x : t) => e</code></td><td><code>Lambda (("x", t), e)</code></td><td>lambda function</td></tr>
<tr><td></td><td>$\mid$</td><td>$e_1\ e_2$</td><td><code>e1 e2</code></td><td><code>Apply (e1, e2)</code></td><td>function application</td></tr>
<tr><td></td><td>$\mid$</td><td>$\mathtt{abort}_\tau(e)$</td><td><code>abort e into t</code></td><td><code>Abort (e, t)</code></td><td>abort</td></tr>
</table>

The specification for propositions can be found below.

		abstract syntax	concrete syntax	ML datatype	description
`prop` p	$::=$	A	`A`	`Atom AA`	atomic proposition
	$\mid$	$\top$	`T`	`True`	truth
	$\mid$	$\bot$	`F`	`False`	falsity
	$\mid$	$p_1 \wedge p_2$	`P1 /\\ P2`	`And (p1, p2)`	conjunction
	$\mid$	$p_1 \vee p_2$	`P1 \\/ P2`	`Or (p1, p2)`	disjunction
	$\mid$	$p_1 \supset p_2$	`P1 => P2`	`Imp (p1, p2)`	implication
	$\mid$	$\neg p$	`~P`	`Not p`	negation

Abstract syntax is the notation we use in the inference rules. Concrete syntax is the notation we use when writing test cases. ML datatype is the underlying implementation of those constructs, which can be found at `slc/SimpleLC.sml`.

Hints and notes:

- One theme of this course is viewing *propositions as types*. This is because we observed a correspondence between propositions and types. For example, conjunction corresponds to product type in STLC; disjunction corresponds to sum type in STLC. For this problem, it may be helpful to implement some helper functions `trans` : `prop -> typ`, which translates a proposition into the corresponding type, and `typecheck` : `exp -> typ -> bool`, which is a type checking algorithm in STLC (based on the typing rules in the appendix). This approach is not the only one, however.

- To test your implementation, run `smlnj -m sources.cm` in the `dist` directory, and use the utility functions in the structure `Top`. Below is a simple example.

```
- Top.check "fn (x :  A) => x" "A => A";
true
val it = () :  unit
```

  More examples can be found in `checker/examples.txt`. You should always come up with your own test cases. You are also encouraged to share your interesting test cases on piazza.

- Note that the language we use here explicitly annotates types for injections, function abstractions, and abort. While these types can in principle be inferred, including them in the syntax of the language simplifies typechecking. Later in the course, we may discuss bidirectional type checking, which allows these annotations to be removed.

- There are two scenarios where the function should return `false`. First, when the proof term is not well typed. For example, checking `fn (x :  A) => x x` should return `false` no matter what the proposition is. Second, when the proof term is well typed, but does not prove the given proposition. For example, checking `fn (x :  unit) => x` against `A => A` should return `false` (the correct type is `unit => unit`).

- You will likely find `Ctx` structure, which defines a dictionary with `string` as the key, helpful for managing contexts of variables and their types. The signature for this structure can be found at `cmlib/dict.sig`.

- Since the `check` function returns a boolean, which only has two possible outcomes, a constant function that always returns `true` will pass many test cases. As such, the score will not be directly proportional to the number of test cases passed.
- While the autograder will grade this problem out of 100, the result will be scaled down (linearly) to 66 points.
- When you submit, run `make`, which generates `handin.zip`. Submit that file to Gradescope.

Typing Rules for STLC

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\text{VAR}) \qquad \frac{}{\Gamma \vdash \langle\rangle : \texttt{unit}}\ (\text{UNIT}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}\ (\text{TUP}) \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}(e) : \tau_1}\ (\text{FST})$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}(e) : \tau_2}\ (\text{SND}) \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{inl}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}\ (\text{INL}) \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{inr}_{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}\ (\text{INR})$$

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{abort}_\tau(e) : \tau}\ (\text{ABORT}) \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}(e; x_1.e_1, x_2.e_2) : \tau}\ (\text{CASE})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2}\ (\text{ABS}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ (\text{APP})$$