CS 599 D1: Assignment 2

Due Wednesday, February 21, 2024

Total: 150 pts

Ankush Das

- This assignment is due on the above date and it must be submitted electronically on Gradescope. Please create an account on Gradescope, if you haven't already done so.
- Please use the template provided on the course webpage to typeset your assignment and please include your name and BU ID in the Author section (above).
- Although it is not recommended, you can submit handwritten answers that are scanned as a PDF and clearly legible.
- You should hand in one file, named as ⟨first-name⟩_⟨last-name⟩_⟨BU-ID⟩_asgn2.pdf containing the solutions to the theory problems below.
- You should also hand in one zip file containing the solutions to the coding assignment.
- You will be provided a tex file, named asgn2.tex. It contains an environment called solution. Please enter your solutions inside these environments.

1 LL1 and λ-Calculus [50 pts]

Let's recall the LL1 language that we have seen in many lectures with a few important changes

$$\begin{array}{rl} \text{Expressions} & e ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \overline{n} \mid \boxed{\underline{n}} \mid \boxed{e + e} \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid x \mid \boxed{e \leq e} \\ \text{Types} & \tau ::= \mathsf{bool} \mid \mathsf{int} \mid \boxed{\mathsf{float}} \end{array}$$

As you can notice, there is a new expression $e_1 \leq e_2$ and a new type for floating-point expressions. We define the $\leq$ operator such that it applies to booleans, integers, and floats. For booleans, we define that false is less than true and then $\leq$ is defined in the standard manner. For integers and floats, the operator $\leq$ is the standard 'less than or equal to' operator. **Note the important caveat that the $\leq$ operator can be used to compare integers with floats and the + operator can be used to add integers and floats** (integers and floats can be on either side of the operator). We also introduce $\underline{n}$ to represent floating-point values.

To prove the theorems in this problem, you can use (without proving) the following lemma.

Lemma 1 (Canonical Forms) *For a well-typed expression e such that $\Gamma \vdash e : \tau$*

- *If τ is* bool*, then $e =$* true *or $e =$* false*.*
- *If τ is* int*, then $e = \overline{n}$ for some n.*
- *If τ is* float*, then $e = \underline{n}$ for some n.*

Problem 1 (34 pts) *Solve the following problems for LL1:*

(4 pts) Define the rules of the type system for LL1. Although not necessary, but try to use as few rules as possible.

(5 pts) Define the rules of the small-step semantics for LL1. To define the semantics, you can use the $\preceq$ and $\neg$ operators to compare integer/float values (similar to $\oplus$ for adding integer/float values).

(5 pts) Define the rules of the big-step semantics for LL1. Again, you can use the $\preceq$ and $\neg$ operators to compare integer/float values.

(10 pts) Is the LL1 language type safe? Either prove the progress and preservation theorems for LL1 or show a counterexample expression e that violates either the progress or the preservation theorem.

[Preservation Theorem]:
If $\Gamma \vdash e : \tau$ and $e \mapsto e'$, then $\Gamma \vdash e' : \tau$.

[Progress Theorem]:
If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ or e value.

(10 pts) Prove the equivalence of the small-step and big-step semantics. To do this, we need to define a new judgment $e \mapsto^* e'$ which is the reflexive transitive closure of $e \mapsto e'$. Formally,

$$\frac{}{e \mapsto^* e}\ \textsc{Refl} \qquad \frac{e_1 \mapsto^* e_2 \qquad e_2 \mapsto^* e_3}{e_1 \mapsto^* e_3}\ \textsc{Trans} \qquad \frac{e_1 \mapsto e_2}{e_1 \mapsto^* e_2}\ \textsc{Step}$$

Write the theorem statement demonstrating the equivalence of the small-step and big-step semantics and then prove it.
***Hint:** Although $e \mapsto^* e'$ is defined as a reflexive transitive closure of $e \mapsto e'$, it might be easier to use this alternative definition*

$$\frac{}{e \mapsto^* e}\ \textsc{Refl} \qquad \frac{e_1 \mapsto e_2 \qquad e_2 \mapsto^* e_3}{e_1 \mapsto^* e_3}\ \textsc{Step}$$

These two definitions are equivalent. You're welcome to prove them, but you don't have to.

Problem 2 (16 pts) *In this problem, fill in the blanks to make the following typing judgments in λ-calculus valid, or briefly explain that it is impossible to do so. This might require defining either (i) an expression with a given type, or (ii) the typing context for an expression, or (iii) the type of a given expression in a typing context, or (iv) a combination of the above. (2 pts each)*

1. $\cdot \vdash \boxed{} : \alpha \to \alpha$
2. $y : \beta \vdash \boxed{} : \alpha \to \beta$
3. $\cdot \vdash \boxed{} : \alpha \to \beta$
4. $\boxed{} \vdash x\ x : \boxed{}$
5. $\cdot \vdash \boxed{} : \alpha \to (\alpha \to \alpha)$
6. $\cdot \vdash \boxed{} : (\alpha \to \alpha) \to \alpha$
7. $\boxed{} \vdash \lambda x.\, x\ (x\ x) : \boxed{}$
8. $\cdot \vdash \lambda f.\, \lambda g.\, \lambda x.\, (f\ x)\ (g\ x) : (\alpha \to \boxed{}) \to (\alpha \to \boxed{}) \to (\alpha \to \boxed{})$

2 System T [60 pts]

In this homework, we will study another popular language called Gödel's System T that supports primitive recursion (but importantly, not general recursion). The language is defined as follows:

Syntax

$$\begin{array}{rl} \text{Expressions} & e ::= \lambda x : \tau.\, e \mid e\ e \mid x \mid \mathsf{zero} \mid \mathsf{succ}(e) \mid \mathsf{natrec}(e\,;\, e\,;\, x.\, y.\, e) \\ \text{Types} & \tau ::= \tau \to \tau \mid \mathsf{nat} \end{array}$$

The language has the standard expressions from λ-calculus except that the λ-expression has the type of argument stated explicitly. In addition, the language natively supports natural numbers (not integers). Natural numbers are defined inductively using (i) zero which defines the natural number 0, and (ii) $\mathsf{succ}(e)$ which defines the successor, i.e., if e denotes the number $\overline{n}$, then $\mathsf{succ}(e)$ denotes $\overline{n+1}$.

Type System

$$\frac{\Gamma, x : \alpha \vdash e : \tau}{\Gamma \vdash \lambda x : \alpha.\, e : \alpha \to \tau}\ \textsc{Lam} \qquad \frac{\Gamma \vdash e_1 : \alpha \to \tau \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{App} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Var}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}}\ \textsc{Zero} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(e) : \mathsf{nat}}\ \textsc{Succ}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathsf{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathsf{natrec}(e\,;\, e_0\,;\, x.\, y.\, e_1) : \tau}\ \textsc{Rec}$$

The rules for λ-calculus are standard. The type of zero is nat and if e has type nat, then $\mathsf{succ}(e)$ also has type nat. Finally, natrec is used to perform primitive recursion on natural numbers (only). The expression $\mathsf{natrec}(e\,;\, e_0\,;\, x.\, y.\, e_1)$ performs recursion on the first argument e which must have type nat. If e is zero, then this expression returns e_0 which has an arbitrary type τ. If e is $\mathsf{succ}(e')$, then we essentially evaluate e_1 where x represents the predecessor e', and y represents the value from the previous recursive call. Therefore, in the presence of $x : \mathsf{nat}$ and $y : \tau$, we need to derive that $e_1 : \tau$.

Semantics

$$\frac{}{\lambda x : \tau.\, e\ \mathsf{value}}\ \lambda\text{-V} \qquad \frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \textsc{App-L} \qquad \frac{e_1\ \mathsf{value} \qquad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \textsc{App-R}$$

$$\frac{e'\ \mathsf{value}}{(\lambda x : \tau.\, e)\ e' \mapsto [e'/x]e}\ \textsc{App-S} \qquad \frac{}{\mathsf{zero}\ \mathsf{value}}\ \textsc{Zero} \qquad \frac{}{\mathsf{succ}(e)\ \mathsf{value}}\ \textsc{Succ}$$

$$\frac{e \mapsto e'}{\mathsf{natrec}(e\,;\, e_0\,;\, x.\, y.\, e_1) \mapsto \mathsf{natrec}(e\,;\, e_0\,;\, x.\, y.\, e_1)}\ \textsc{Rec-E} \qquad \frac{}{\mathsf{natrec}(\mathsf{zero}\,;\, e_0\,;\, x.\, y.\, e_1) \mapsto e_0}\ \textsc{Rec-Z}$$

$$\frac{}{\mathsf{natrec}(\mathsf{succ}(e)\,;\, e_0\,;\, x.\, y.\, e_1) \mapsto [e/x,\ \mathsf{natrec}(e\,;\, e_0\,;\, x.\, y.\, e_1)/y]e_1}\ \textsc{Rec-S}$$

The rules of λ-calculus are standard. zero and $\mathsf{succ}(e)$ are defined to be values. There is a standard rule Rec-E for evaluating the argument to natrec. If the argument is 0, the natrec expression simply steps to e_0 as demonstrated by the Rec-Z rule. Finally, if the argument is $\mathsf{succ}(e)$, then e is substituted for x and the recursive call $\mathsf{natrec}(e\,;\, e_0\,;\, x.\, y.\, e_1)$ is substituted for y in e_1. **Try to write some examples of e_0 and e_1 to see how they evaluate.**

For your convenience, we will state (but not prove) a canonical forms lemma.

Lemma 2 (Canonical Forms) *If $\Gamma \vdash e : \tau$, then*

- *If τ is nat, then $e = \mathsf{zero}$ or $e = \mathsf{succ}(e')$ for some e' such that $\Gamma \vdash e' : \mathsf{nat}$.*
- *If τ is $\tau_1 \to \tau_2$, then $e = \lambda x : \tau_1.\, e'$ for some e' such that $\Gamma, x : \tau_1 \vdash e' : \tau_2$.*

2.1 Termination

Unlike general recursive programming languages like OCaml and Rust, System T has the valuable property that all programs written in this language terminate, i.e., evaluate to a value in a finite number of steps. Your task is to prove this fact using Tait's reducibility method. The theorem we want to prove is the following:

Theorem 1 (Normalization) *If $\cdot \vdash e : \tau$, then there exists v such that v* value *and $e \mapsto^* v$, where $\mapsto^*$ is the reflexive transitive closure of $\mapsto$.*

We might hope to prove this theorem directly by induction on the typing judgment. However, this approach is insufficient. The case for the application rule (App) is demonstrative.

$$\frac{\Gamma \vdash e_1 : \alpha \to \tau \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{App}$$

In this case, our induction hypotheses tells us that $e_1 \mapsto^* v_1$ and $e_2 \mapsto^* v_2$ for some values v_1 and v_2. By preservation and the appropriate canonical forms lemma, we know that $v_1 = \lambda x : \alpha.\, e'$ for some e'. It also follows that $e_1\ e_2 \mapsto^* v_1\ e_2 \mapsto [e_2/x]e'$. Unfortunately, we are now stuck, as we have no information about the behavior of $[e_2/x]e'$.

We will solve this by generalizing, proving a stronger statement which gives us more information as an induction hypothesis. Specifically, we will define a *reducibility predicate* $\mathsf{Red}_\tau(e)$ and prove the following theorem.

Theorem 2 *If $\cdot \vdash e : \tau$, then $\mathsf{Red}_\tau(e)$.*

Since we'll define Red_τ such that $\mathsf{Red}_\tau(e)$ implies the existence of v value with $e \mapsto^* v$, this theorem will imply normalization as a corollary. The definition will go by structural induction on the type τ, which makes Red_τ what is called a *logical relation.* (In particular, it is a *unary* logical relation; we will encounter *binary* logical relations, such as logical equivalence $e \sim_\tau e'$, later in the course.) Actually, we will prove an even more general theorem in order to account for open terms; to state it concisely, we first want to define some notation for substitutions.

Definition 1 *A substitution $\gamma = \{x_1 \hookrightarrow e_1, \ldots, x_n \hookrightarrow e_n\}$ is a finite mapping from variables to terms. Given an expression e, we write $\gamma(e)$ for the expression $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$, that is, the simultaneous substitution in e of each expression e_i for its corresponding variable x_i. For γ as above, we define $\gamma \Vdash \Gamma$ to mean that $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ for some $\tau_1, \ldots, \tau_n$ such that $\mathsf{Red}_{\tau_i}(e_i)$ holds for $1 \le i \le n$.*

Now we state the theorem we will actually prove:

Theorem 3 *If $\Gamma \vdash e : \tau$ and $\gamma \Vdash \Gamma$ then $\mathsf{Red}_\tau(\gamma(e))$.*

Theorem 2 follows as the special case where $\Gamma = \cdot$ and $\gamma = \langle\rangle$. Finally, we define the predicate Red_τ by structural induction on τ:

- $\mathsf{Red}_{\tau_1 \to \tau_2}(e)$ holds if
  1. $\cdot \vdash e : \tau_1 \to \tau_2$,
  2. there exists v value such that $e \mapsto^* v$, and
  3. for any e' such that $\mathsf{Red}_{\tau_1}(e')$, we have $\mathsf{Red}_{\tau_2}(ee')$.
- $\mathsf{Red}_{\mathsf{nat}}(e)$ holds if
  1. $\cdot \vdash e : \mathsf{nat}$,
  2. there exists v value such that $e \mapsto^* v$, and
  3. $v \downarrow$, where $v \downarrow$ is a judgment defined by

$$\frac{}{\mathsf{zero} \downarrow}\ \downarrow\text{-Z} \qquad\qquad \frac{e \mapsto^* v \qquad v\ \mathsf{value} \qquad v \downarrow}{\mathsf{succ}(e) \downarrow}\ \downarrow\text{-S}$$

Note that $\mathsf{Red}_{\tau_1 \to \tau_2}(e)$ is defined in terms of Red at the structurally smaller types τ_1 and τ_2, so the definition is well-founded. To get you started on the proof, and to see how this definition succeeds where the previous attempt failed, here is the App case:

- Case App: We have $\Gamma \vdash e_1\ e_2 : \tau$ with $\Gamma \vdash e_1 : \alpha \to \tau$ and $\Gamma \vdash e_2 : \alpha$ for some α. Per the theorem statement, we assume we are given $\gamma \Vdash \Gamma$ and want to prove that $\mathsf{Red}_\tau(\gamma(e_1\ e_2))$. By definition of substitution, we have that $\gamma(e_1\ e_2) = \gamma(e_1)\ \gamma(e_2)$. Moreover, our induction hypotheses tell us that $\mathsf{Red}_{\alpha\to\tau}(\gamma(e_1))$ and $\mathsf{Red}_\alpha(\gamma(e_2))$. From condition 3 in the definition of $\mathsf{Red}_{\alpha\to\tau}$, we know that for any e' with $\mathsf{Red}_\alpha(e')$ we have $\mathsf{Red}_\tau(\gamma(e_1)e')$. Taking $e' = \gamma(e_2)$ thus gives our goal.

With the right definition Red in hand, the App case follows almost trivially. On the other hand, the Lam case becomes more difficult. In general, though, proving the theorem is the easy part of a logical relations argument – the hard part is choosing the right theorem to prove.

To complete the proof, you'll need the following lemma.

Lemma 3 (Closure under Head Expansion) *If* $\mathsf{Red}_\tau(e')$, $\cdot \vdash e : \tau$ *and* $e \mapsto e'$, *then* $\mathsf{Red}_\tau(e)$.

Problem 3 (10 pts) *Prove closure under head expansion.*

With the help of Preservation, closure under head expansion extends to apply when $e \mapsto^* e'$ in multiple steps (you can use this without proof).

Problem 4 (30 pts) *Prove the remaining cases of Theorem 3. You may state (without proof) lemmas about substitution, but be sure to check that they are actually true.*

2.2 Programming in System T

Next, we will get some programming experience in System T.

Problem 5 (20 pts) *Define the following functions in System T. For each definition below, briefly explain the intuition behind your answer (4 pts each). You can define and use helper functions to solve these problems.*

1. *Define* mult, *where* mult $\overline{m}\ \overline{n} \mapsto^* \overline{m \otimes n}$.
2. *Define* minus, *where* minus $\overline{m}\ \overline{n} \mapsto^* \overline{m - n}$ *if* $\overline{m} > \overline{n}$. *It should produce 0 otherwise.*
3. *Define* leq, *where* leq $\overline{m}\ \overline{n} \mapsto^*$ succ(zero) *if* $\overline{m} \le \overline{n}$ *and* leq $\overline{m}\ \overline{n} \mapsto^*$ zero *otherwise.*
4. *Define* mod *where* mod $\overline{m}\ \overline{n} = \overline{m\ \mathsf{mod}\ n}$. *You may pick appropriate defaults when* $n = 0$.
5. *Define* cube *where* cube $\overline{n} = \overline{n \otimes n \otimes n}$.

3 Coding Assignment [40 pts]

The last set of problems in this assignment involve coding in your favorite programming language. Since you are free to choose your own language, please follow the guidelines below to make your submission can be accepted by the instructor.

- In this set of problems, you will be required to define several types and functions.
- Please make your submission is a zip file that contains the code file that defines the required functions and types.
- Your zip file must include a separate readme file that contains instructions for installing and executing the file(s) in your submission.
- You're welcome to modularize your code into multiple files (but you don't need to). If you do create several files, please indicate in your instructions which file contains the functions and types for each problem.

- Begin coding!

We will implement a slightly simplified version of the LL1 language in this section.

$$\begin{array}{rl} \text{Expressions} & e ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \overline{n} \mid e + e \mid \mathsf{let}\ x : \tau = e\ \mathsf{in}\ e \mid x \\ \text{Types} & \tau ::= \mathsf{bool} \mid \mathsf{int} \end{array}$$

You will notice that the let expression now has an explicit type annotation for the variable x. This will make it easier to implement the type checker of the language. We will remove this simplification in the future.

Problem 6 (5 pts) *Define a type called*

(1 pts) `tp` *for types in LL1,*

(2 pts) `exp` *for expressions in LL1, and*

(2 pts) `value` *for values in LL1*

Problem 7 (10 pts) *Define a function called* `typecheck` *with the following signature:*

```
typecheck: context -> exp -> tp -> bool
```

This function essentially implements the typing rules we've defined in the lectures using the judgment $\Gamma \vdash e : \tau$. *It takes a context, an expression, and a type as input and returns* `true` *if the expression has that type, and returns* `false` *otherwise. You're welcome to choose your own type for* `context` *that stores the type of all the variables in the context.*

Problem 8 (10 pts) *Define a function called* `step` *with the following signature:*

```
step: exp -> result
```

This function implements one step of evaluation as defined in the small-step semantics judgment $e \mapsto e'$ *For this function, define the appropriate return type (called* `result` *here) which can either be an expression or a value. To define this function, you will also need to define an appropriate substitution function that substitutes a value for a variable in an expression. This function will (likely) have the following signature:*

```
substitute: value -> variable -> exp -> exp
```

You will observe that some of the cases in this function are impossible (you can raise an exception in these cases if you like). Which of the cases are impossible and why? Write this in comments next to the impossible cases.

Problem 9 (5 pts) *Define a function called* `eval` *with the following signature:*

```
eval: exp -> value
```

This function essentially recursively calls `step` *until it returns a value. Also, define 3 printing functions with the following signatures*

```
string_of_exp: exp -> string
string_of_value: value -> string
string_of_tp: tp -> string
```

The first function essentially converts an expression into a string. This string should look exactly as the grammar written in the start of this section. The second function does the same for values and the third function does the same for types. Use these three functions in the body of `eval` *to print out all the intermediate expressions and the final value of evaluation.*

Problem 10 (5 pts) *Define a function called* `big_step` *with the following signature:*

```
big_step: exp -> value
```

This function implements the big-step semantics judgment $e \Downarrow v$.

Problem 11 (5 pts) *Write as many tests as you can for all the functions defined above. In particular,*

1. *Write at least 5 examples of closed ill-typed LL1 programs. Test that* `typecheck` *returns* `false` *for each of these examples.*
2. *Write at least 10 examples of closed well-typed LL1 programs. Test that* `typecheck` *returns* `true` *for each of these examples. Also, test that* `eval` *and* `big_step` *return the same value for all these examples.*

Here are some examples to help you get started.

- let x : int $= y$ in $x + y$
- if 3 then true else false
- if true then true else $5 + 8$
- if true then 3 else $5 + 8$
- let x : int $= 5$ in let $y = x + y$ in y
- let x : int $=$ true in if x then let $y = 1$ in $y + y$ else false
- let x : int $= 5$ in let $y = x + x$ in $y + (y + x)$