Homework 2023

This homework must be performed by group, with the same group you are using for programming assignments (hence, from 1 to 3 students).

The homework *must* be written *by hand*. Not using LaTeX or any other software.

The original copy of your solution must be handed back before Wednesday November 15, 2023, 6:15pm, to the office BC 359. If there is no one, there will be a box outside the door. After that deadline, the usual penalties to your grade apply.

Be very precise in your proofs:

- State what kind of induction you are using, if any.
- State the possible cases of a case analysis as clear subtitles (underline them, for example).
- When you use a theorem or a lemma, mention which one.
- If you need an additional lemma, give it a number, prove it, and refer to it when you use it.

Exercise 1 : Hacking with the untyped call-by-value lambda calculus (10 points)

In this exercise, you have to implement some operations for Church encoding of signed numbers. One approach for encoding signed numbers in lambda calculus is to use a Church pair, containing Church numerals representing a positive and a negative value. The signed number value is the difference between the two Church numerals. As an example 3 is represented as follows:

$$\texttt{pair c}_3\ \texttt{c}_0$$

where $\texttt{c}_3$ and $\texttt{c}_0$ are Church encoding of 3 and 0, respectively. It could also be represented as

$$\texttt{pair c}_5\ \texttt{c}_2$$

A negative number such as -3 is represented, for example, as follows:

$$\texttt{pair c}_0\ \texttt{c}_3$$

or

$$\texttt{pair c}_1\ \texttt{c}_4$$

A Church encoded natural number is converted to a Church encoded signed number in the following way:

$$\lambda\texttt{n.pair n c}_0$$

The complete list of predefined operations can be found in the appendix, and only those operations may be used in the exercise. (Note that the fixpoint combinator `fix` is not listed in the appendix and may therefore *not* be used.) In addition, you are allowed to use any helper functions you define yourself (including the solutions to previous questions).

An incorrect solution that can be made correct by adding parentheses is worth half of the points.

Define the following operations on signed numbers:

1. (2 points) The function (`negs x`) which returns the negation of the given signed number `x`. The result should be a Church-encoded signed number.

2. (2 points) The function (`adds x y`) which returns the addition of two signed numbers `x` and `y`. The result should be a Church-encoded signed number.

3. (2 points) The function (`subs x y`) which returns the subtraction of two signed numbers `x` and `y`. The result should be a Church-encoded signed number.

4. (4 points) The function (`mults x y`) which returns the multiplication of two signed numbers `x` and `y`. The result should be a Church-encoded signed number.

Exercise 2 : Affine lambda calculus (10 points)

The affine lambda calculus is a variation of the lambda calculus with call-by-value semantics in which, for every sub-term $\texttt{t}$ of the form $\lambda\texttt{x.s}$, the bound variable $\texttt{x}$ appears at most once in the body $\texttt{s}$. For example: $\lambda\texttt{x.x}$ and $\lambda\texttt{x.}\lambda\texttt{y.x y}$ are affine, while $\lambda\texttt{x.x x}$ is not.

Formally, we define affine($\texttt{t}$) as the smallest predicate on terms that satisfies the following rules:

$$\text{affine}(\texttt{x}) \qquad \text{(A-VAR)} \qquad\qquad \frac{\text{affine}(\texttt{t}_1)}{\text{affine}(\lambda\texttt{x.t}_1)} \qquad \text{(A-ABS)}$$

$$\frac{\text{affine}(\texttt{t}_1) \qquad \text{affine}(\texttt{t}_2) \qquad \text{FV}(\texttt{t}_1) \cap \text{FV}(\texttt{t}_2) = \emptyset}{\text{affine}(\texttt{t}_1\ \texttt{t}_2)} \qquad \text{(A-APP)}$$

Prove that all programs in this language terminate. In other words, show that for any term $\texttt{t}$ such that affine($\texttt{t}$), $\texttt{t}$ eventually reaches a normal form. *Hint:* Think about what happens to the number of lambda abstractions after one reduction step.

You may use the two lemmas below without proving them.

Lemma 1. *For any pair of terms $\texttt{s}$, $\texttt{t}$ and variable $\texttt{x}$, if $\texttt{x} \notin \text{FV}(\texttt{s})$ then $[\texttt{x} \mapsto \texttt{t}]\texttt{s} = \texttt{s}$.*

Lemma 2. *If* affine($\texttt{t}$) *and $\texttt{t} \longrightarrow \texttt{t}'$, then* affine($\texttt{t}'$).

Exercise 3 : Adding the update operation to labeled records (10 points)

In this exercise, we build an extension to a call-by-value simply typed lambda calculus with labeled records. We would like to enhance the original calculus introduced in TAPL (Section 11.8) with a construct that would make it easy to create slightly different records from existing ones. The syntax of the new operation is shown in the EBNF form:

$\mathtt{t}$	::=	...	**terms**
	|	$\mathtt{t}_1$ `update l with` $\mathtt{t}_2$	updates field `l` of a record $\mathtt{t}_1$ with $\mathtt{t}_2$

`update` takes a record, a field label that is already in the record, and a new value of the type that matches the existing field. It then "updates" the record (returns a new one) with the new value of that field. For example:

$$\{\mathtt{a=1}, \mathtt{b=false}\}\ \mathtt{update\ b\ with\ true} \rightarrow \{\mathtt{a=1}, \mathtt{b=true}\}$$

Here is the typing rule for `update` terms:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i : \mathtt{T}_i^{i \in 1..n}\} \qquad j \in 1..n \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_j}{\Gamma \vdash \mathtt{t}_1\ \mathtt{update\ l}_j\ \mathtt{with\ t}_2 : \{\mathtt{l}_i : \mathtt{T}_i^{i \in 1..n}\}} \qquad \text{(T-Update)}$$

Your task for this assignment is as follows:

1. (4 points) Introduce new evaluation rules that follow from the given syntactic extension and the behavior described above. They must be defined so that the whole system is sound, i.e., it must be possible to prove progress and preservation. Defining useless rules (rules that do not contribute to the progress proof) will be penalized.

2. (6 points) Prove soundness of your system by detailing new cases of inductive proofs of progress and preservation. You need only discuss the cases that result from the typing rule introduced above and/or the evaluation rules that you introduced. If needed, you may use (without proving them) the standard lemmas of inversion of typing, canonical forms, substitution lemma, and uniqueness of types for labeled records.

Exercise 4 : Dynamic (10 points)

In this exercise we extend the Simply Typed Lambda Calculus with a `Dynamic` type. The `Dynamic` type allows us to escape the static nature of STLC and create some interesting constructs.

For instance, for STLC with arithmetic expressions, $(\lambda \texttt{x:Nat.0})\ (\lambda \texttt{y:Nat.0})$ is not typeable, even though the variable `x` is not used within the body of the value abstraction. With `Dynamic`, we would be able to write a well-typed term like

$$(\lambda \texttt{x:Dynamic.0})\ (\texttt{dynamic}\ (\lambda \texttt{y:Nat.0})\ \texttt{as}\ \texttt{Nat} \to \texttt{Nat})$$

The `dynamic` construct would therefore *package* a value and its type. *Unpacking* dynamic values is only possible through the `typecase` construct. It matches on the underlying type of the `Dynamic` (similarly to Sums which were defined during the lectures). Below we give a tentative formalization of this extension.

`t`	::=	...	**terms**
	|	`dynamic t as T`	pack dynamic value
	|	`typecase t match {case x:T ⇒ t else t}`	unpack dynamic value
`v`	::=	...	**values**
	|	`dynamic v as T`	dynamic value
`T`	::=	...	**types**
	|	`Dynamic`	Dynamic type

Typing and evaluation rules:

$$\Gamma \vdash \texttt{dynamic}\ \texttt{t}_1\ \texttt{as}\ \texttt{T}_1\texttt{:Dynamic} \qquad \text{(T-Dynamic)}$$

$$\frac{\Gamma \vdash \texttt{t}_1\texttt{:Dynamic} \qquad \Gamma, \texttt{x:U} \vdash \texttt{t}_2\texttt{:T} \qquad \Gamma \vdash \texttt{t}_3\texttt{:T}}{\Gamma \vdash \texttt{typecase}\ \texttt{t}_1\ \texttt{match}\ \{\texttt{case x:U} \Rightarrow \texttt{t}_2\ \texttt{else}\ \texttt{t}_3\}\texttt{:T}} \qquad \text{(T-Typecase)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{dynamic}\ \texttt{t}_1\ \texttt{as}\ \texttt{T}_1 \longrightarrow \texttt{dynamic}\ \texttt{t}_1'\ \texttt{as}\ \texttt{T}_1} \qquad \text{(E-Dynamic)}$$

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{\texttt{typecase}\ \texttt{t}_1\ \texttt{match}\ \{\texttt{case x:U} \Rightarrow \texttt{t}_2\ \texttt{else}\ \texttt{t}_3\} \longrightarrow \texttt{typecase}\ \texttt{t}_1'\ \texttt{match}\ \{\texttt{case x:U} \Rightarrow \texttt{t}_2\ \texttt{else}\ \texttt{t}_3\}} \qquad \text{(E-Typecase)}$$

$$\texttt{typecase}\ (\texttt{dynamic}\ \texttt{v}_{11}\ \texttt{as}\ \texttt{U})\ \texttt{match}\ \{\texttt{case x:U} \Rightarrow \texttt{t}_2\ \texttt{else}\ \texttt{t}_3\} \longrightarrow [\texttt{x} \to \texttt{v}_{11}]\,\texttt{t}_2 \qquad \text{(E-Typecase-Match)}$$

$$\frac{\texttt{T} \neq \texttt{U}}{\texttt{typecase}\ (\texttt{dynamic}\ \texttt{v}_{11}\ \texttt{as}\ \texttt{T})\ \texttt{match}\ \{\texttt{case x:U} \Rightarrow \texttt{t}_2\ \texttt{else}\ \texttt{t}_3\} \longrightarrow \texttt{t}_3} \qquad \text{(E-Typecase-Else)}$$

For this exercise you have to:

- (2 points) Show that progress and preservation **do not** hold by giving a brief example of each violation.

- (1 point) Make the extension sound by *fixing 1 typing rule* (no evaluation rule, and no more than 1 typing rule, no removal or addition of any rule). The resulting system must still allow "reasonable" programs to type-check (for example, adding a `false` premise to a rule to make the system unusable is *not* a valid solution). In particular, every evaluation rule must still be necessary to be able to prove progress.
- (7 points) Prove soundness of your extension (progress and preservation) by detailing new cases for the inductive proof. You need to discuss only the cases that result from the evaluation and typing rules introduced above. If needed, you may use (without proving them) the standard lemmas of *inversion of typing*, *canonical forms*, *substitution lemma*, and *uniqueness of types*.

For reference: **Predefined Lambda Terms**

Predefined lambda terms that can be used as-is in the first exercise:

$$
\begin{array}{lcl}
\texttt{unit} & = & \lambda\texttt{x.x} \\
 & & \\
\texttt{tru} & = & \lambda\texttt{t.}\lambda\texttt{f.t} \\
\texttt{fls} & = & \lambda\texttt{t.}\ \lambda\texttt{f.f} \\
\texttt{iszro} & = & \lambda\texttt{m.m}\ (\lambda\texttt{x.fls})\ \texttt{tru} \\
\texttt{test} & = & \lambda\texttt{b.}\lambda\texttt{t.}\lambda\texttt{f.b t f unit} \\
 & & \\
\texttt{pair} & = & \lambda\texttt{f.}\lambda\texttt{s.}\lambda\texttt{b.b f s} \\
\texttt{fst} & = & \lambda\texttt{p.p tru} \\
\texttt{snd} & = & \lambda\texttt{p.p fls} \\
 & & \\
\texttt{c}_0 & = & \lambda\texttt{s.}\lambda\texttt{z.z} \\
\texttt{c}_1 & = & \lambda\texttt{s.}\lambda\texttt{z.s z} \\
\texttt{scc} & = & \lambda\texttt{n.}\lambda\texttt{s.}\lambda\texttt{z.s}\ (\texttt{n s z}) \\
\texttt{plus} & = & \lambda\texttt{m.}\lambda\texttt{n.}\lambda\texttt{s.}\lambda\texttt{z.m s}\ (\texttt{n s z}) \\
\texttt{times} & = & \lambda\texttt{m.}\lambda\texttt{n.m}\ (\texttt{plus n})\ \texttt{c}_0 \\
 & & \\
\texttt{zz} & = & \texttt{pair c}_0\ \texttt{c}_0 \\
\texttt{ss} & = & \lambda\texttt{p.pair}\ (\texttt{snd p})\ (\texttt{scc}\ (\texttt{snd p})) \\
\texttt{prd} & = & \lambda\texttt{m.fst}\ (\texttt{m ss zz})
\end{array}
$$