1 Hacking with the untyped call-by-value lambda calculus

Pierce explains Church encodings for booleans, numerals, and operations on them in (TAPL book s. 5.2 p. 58). We would like to define some more advanced functions using only the basic operations $scc, plus, prd, times, iszro, fix$ and the constants. The complete list of predefined operations can be found in the appendix, and only these operations can be used in the exercise. Define the following operations on non-negative integers:

1. The greater equal operation *geq* ($\geq$)

2. The greater than operation *gr* ($>$)

3. The modulo operation *mod* (e.g. $mod\ c_5\ c_3 = c_2$)

4. The Ackermann function *ack* using the basic operations and operations defined in this exercise. The Ackermann function is defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

2 Uniqueness of terms after reductions

For this exercise assume we will be working with a simple language defined using the following algebraic datatype in `Scala`:

```
abstract class Base
case object Empty extends Base
case class Unary(i: Base) extends Base
case class Binary(i: Base, j: Base) extends Base
```

Let $\rightarrow^{\beta}$ be a reduction defined by the following reductions:

$$\texttt{Binary(Empty, x)} \rightarrow \texttt{x} \tag{β_1}$$

$$\texttt{Binary(x, y)} \rightarrow \texttt{Binary(y, x)} \tag{β_2}$$

and congruence rules:

$$\frac{\texttt{x} \rightarrow \texttt{x'}}{\texttt{Unary(x)} \rightarrow \texttt{Unary(x')}} \tag{β_3}$$

$$\frac{\texttt{x} \rightarrow \texttt{x'}}{\texttt{Binary(x, y)} \rightarrow \texttt{Binary(x', y)}} \tag{β_4}$$

$$\frac{\texttt{y} \rightarrow \texttt{y'}}{\texttt{Binary(x, y)} \rightarrow \texttt{Binary(x, y')}} \tag{β_5}$$

Prove that for any elements u, v, w of type `Base` we have

$$u \rightarrow^{\beta} v \ \wedge \ u \rightarrow^{\beta} w \ \Rightarrow \exists x.(v \rightarrow^{*\beta} x \ \wedge \ w \rightarrow^{*\beta} x)$$

where $\rightarrow^{*\beta}$ refers to 0 or more $\rightarrow^{\beta}$ reductions. Your solution has to have a clear explanation for each step in your proof.

3 The call-by-value simply typed lambda calculus with returns

Consider a variant of the call-by-value simply typed lambda calculus specified in the appendix extended to support a new language construct: `return` t, which immediately returns a given term t from an **enclosing** lambda, disregarding any potential further computation typically needed for call-by-value evaluation rules.

The grammar of the extension is defined as follows. We distinguish top-level terms (tt) and nested terms (nt) to make sure that `return` t can only appear inside lambdas:

$$
\begin{array}{lll@{\qquad}l}
v & ::= & \lambda x : \texttt{T} \,.\, nt \mid bv & (values) \\
bv & ::= & \texttt{true} \mid \texttt{false} & (boolean\ values) \\
nt & ::= & x \mid v \mid nt\ nt \mid \texttt{return}\ nt & (nested\ terms) \\
tt & ::= & x \mid v \mid tt\ tt & (top-level\ terms) \\
t & ::= & nt \mid tt & (terms) \\
p & ::= & tt & (programs) \\
T & ::= & \texttt{Bool} \mid \texttt{T} \to \texttt{T} & (types)
\end{array}
$$

In this exercise, you are to adjust the existing evaluation and typing rules, so that they correctly and comprehensively describe the semantics of the extension. More precisely, your task is two-fold:

1. Extend the evaluation rules (by adding new rules and/or changing existing ones) to express the early return semantics provided by return. Identify the evaluation strategy used by the specification and make sure that your extension adheres to it.
2. Extend the typing rules (by adding new rules and/or changing existing ones) to guarantee that types of values returned via return and via normal means are coherent. Make sure that progress and preservation conditions hold for your extension (you don't have to prove that formally, but your grade will be reduced if your extension ends up being unsafe).

   Hint: In addition to the immediate type of a term, you also need to keep track of the types of returned terms inside that term. For example, instead of the regular typing judgment $\Gamma \vdash t : \texttt{T}$, you can use the $\Gamma \vdash t : \texttt{T} \mid \texttt{R}$, where $\texttt{R}$ is a set of types of terms, i.e. $\{T_1, ...T_n\}$, that can be returned from `t`.

Before you begin, think carefully about the following simple term: $\lambda x : \texttt{Bool}. (\texttt{return}\,\texttt{true})\ x$. Intuitively, it makes sense. Once this lambda is applied, it is going to evaluate to `true`, regardless of the input. Now, which typing rules would be used to type this term, so that it is accepted by our language? In particular, what type or types need to be assigned to `return true`?

4 Closed terms

We recall that a term t is closed if it contains no free variables. With that definition in mind prove the following property for the call-by-value untyped lambda calculus (for reference provided in Appendix 1).

Theorem: If t is closed, and $t \longrightarrow t'$ then t' is closed as well.

5 Dynamic

In this exercise we extended 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, $(\lambda x : Nat.0)(\lambda y : Nat.0)$ is not typeable even though variable x is not used within the body of the value abstraction. With `Dynamic`, we would be able to write a term like $(\lambda x : Dynamic.\ 0)\ (dynamic\ (\lambda y : Nat.\ 0)\ as\ Nat \to Nat)$ though.
The `dynamic` construct would therefore *package* a value and its type. *Unpacking* dynamic values is only possible through the `typecase` construct that matches on the underlying type of `Dynamic` (similarly to `Sums` which were defined during the lectures).
Below we give a tentative formalization of this extension.

$$\begin{array}{rcll}
t & ::= & \ldots & \textbf{terms} \\
 & | & \texttt{dynamic}\ t\ \texttt{as}\ T & \text{pack dynamic value} \\
 & | & \texttt{typecase}\ t\ \texttt{match}\ \{\ \texttt{case}\ x_i^{i\in 1..n} : T_i \Rightarrow t_i\ \} & \text{unpack dynamic value} \\
 & & & \\
v & ::= & \ldots & \textbf{values} \\
 & | & \texttt{dynamic}\ v\ \texttt{as T} & \text{dynamic value} \\
 & & & \\
T & ::= & \ldots & \textbf{types} \\
 & | & \texttt{Dynamic} & \text{Dynamic type}
\end{array}$$

Typing and evaluation rules:

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

$$\frac{\Gamma \vdash \texttt{t} : \texttt{Dynamic} \qquad \Gamma, \texttt{x}_i : \texttt{T}_i \vdash \texttt{t}_i : \texttt{T}}{\Gamma \vdash \texttt{typecase t match}\ \{\texttt{case x}_i^{i\in 1..n} : \texttt{T}_i\ \Rightarrow \texttt{t}_i\} : \texttt{T}} \qquad \text{(T-TYPECASE)}$$

$$\frac{k \in 1..n \wedge \texttt{T}_k = \texttt{T} \qquad for\ each\ j, j < k\ \Rightarrow\ \texttt{T}_j\ \neq \texttt{T}}{\texttt{typecase (dynamic v as T) match}\ \{\texttt{case x}_i^{i\in 1..n} : \texttt{T}_i\ \Rightarrow \texttt{t}_i\} \to [\texttt{x}_k \to \texttt{v}]\,\texttt{t}_k} \qquad \text{(E-TYPECASE-DYNAMIC)}$$

$$\frac{\texttt{t} \to \texttt{t}'}{\texttt{dynamic t as T} \to \texttt{dynamic t}'\ \texttt{as T}} \qquad \text{(E-DYNAMIC)}$$

$$\frac{\texttt{t} \to \texttt{t}'}{\begin{array}{c}\texttt{typecase t match}\ \{\ \texttt{case x}_i^{i\in 1..n} : \texttt{T}_i\ \Rightarrow \texttt{t}_i\} \to \\ \texttt{typecase t}'\ \texttt{match}\ \{\ \texttt{case x}_i^{i\in 1..n} : \texttt{T}_i\ \Rightarrow \texttt{t}_i\}\end{array}} \qquad \text{(E-TYPECASE)}$$

For this exercise you have to:

- Show that progress and preservation **do not** hold by giving a brief example of each violation.
- Make the extension sound (by making minimal number of changes to the grammar, typing and evaluation rules). Note that having a reduction rule which reduces a term to itself is not a desirable semantics for this exercise.

- Prove progress of your extension by detailing new cases for the inductive proof. You need to discuss only the cases that result from the evaluation and typing rules that were introduced, and, if needed, you may use (without proving them) the standard lemmas of *inversion of typing*, *canonical forms*, and *uniqueness of types*.
  Although you do not need to prove preservation you have to make sure that your changes in the previous step make it succeed.

For reference: **predefined lambda terms**

Predefined lambda terms that can be used as-is in “Hacking with the untyped call-by-value lambda calculus”:

`tru` =	$\lambda t.\ \lambda f.\ t$
`fls` =	$\lambda t.\ \lambda f.\ f$
`iszro` =	$\lambda m.\ m\ (\lambda x.\ fls)\ tru$
`pair` =	$\lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s$
`fst` =	$\lambda p.\ p\ tru$
`snd` =	$\lambda p.\ p\ fls$
$\mathtt{c}_0$ =	$\lambda s.\ \lambda z.\ z$
$\mathtt{c}_1$ =	$\lambda s.\ \lambda z.\ s\ z$
`scc` =	$\lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$
`plus` =	$\lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$
`times` =	$\lambda m.\ \lambda n.\ m\ (plus\ n)\ c_0$
`zz` =	$pair\ c_0\ c_0$
`ss` =	$\lambda p.\ pair\ (snd\ p)\ (scc\ (snd\ p))$
`prd` =	$\lambda m.\ fst\ (m\ ss\ zz)$
`fix` =	$\lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$

The call-by-value simply typed lambda calculus

The complete reference of the variant of simply typed lambda calculus (with *Bool* ground type representing the type of values *true* and *false*) used in “The call-by-value simply typed lambda calculus with returns” and “Dynamic” is as follows:

$$\begin{array}{ll}
v ::= \lambda x : T.\ t \mid bv & (\textit{values}) \\
bv ::= \texttt{true} \mid \texttt{false} & (\textit{boolean values}) \\
t ::= x \mid v \mid t\ t & (\textit{terms}) \\
p ::= t & (\textit{programs}) \\
T ::= \texttt{Bool} \mid T \to T & (\textit{types})
\end{array}$$

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_1)\ v_2 \longrightarrow [x \to v_2]t_1 \qquad \text{(E-AppAbs)}$$

Typing rules:

$$\frac{x\ \texttt{:}\ \texttt{T} \in \Gamma}{\Gamma \vdash x\ \texttt{:}\ \texttt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\ x\ \texttt{:}\ \texttt{T}_1 \vdash t_2\ \texttt{:}\ \texttt{T}_2}{\Gamma \vdash (\lambda x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_2)\ \texttt{:}\ \texttt{T}_1 \to \texttt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1\ \texttt{:}\ \texttt{T}_1 \to \texttt{T}_2 \quad \Gamma \vdash t_2\ \texttt{:}\ \texttt{T}_1}{\Gamma \vdash t_1\ t_2\ \texttt{:}\ \texttt{T}_2} \qquad \text{(T-App)}$$

$$\frac{}{\Gamma \vdash \texttt{true : Bool}} \qquad \text{(T-False)}$$

$$\frac{}{\Gamma \vdash \texttt{false : Bool}} \qquad \text{(T-True)}$$