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 $scc, plus, prd, times, iszro, test, fix$ 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* ($\geq$)

```
geq = λx. λy. iszro (x prd y)
```

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

```
gr = λx. λy. (iszro (y prd x)) fls tru
```

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

```
mod =
  fix λme. λx. λy.
    test (geq x y)
    (λthen. (me (y prd x) y))
    (λelse. x)
```

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

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

```
ack =
  fix λme. λx. λy.
    test (iszro x)
    (λthen. (scc y))
    (λelse.
      test (iszro y)
      (λthen. (me (prd x) c1))
      (λelse. (me (prd x) (me x (prd y)))))
```

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.

Hint: We suggest proving the above statement using induction on the structure of `u`.

Solution: If $v = w$ then we are done since $x = v = w$. Therefore it remains to prove the case when $v \neq w$. We prove it using induction on the structure of `u`.

- Case $\underline{u = \mathit{Empty}}$. No reduction applies. Done.
- Case $\underline{u = \mathit{Unary}(u_1)}$. Then $u \rightarrow^{\beta_3} v \ \wedge \ u \rightarrow^{\beta_3} w$. Additionally v and w are of the form $\mathit{Unary}(v_1)$ and $\mathit{Unary}(w_1)$, respectively, because of $u_1 \rightarrow^{\beta} v_1$ and $u_1 \rightarrow^{\beta} w_1$. By IH we have that $\exists x_1.(v_1 \rightarrow^{\beta *} x_1 \wedge w_1 \rightarrow^{\beta *} x_1)$. Hence, for $x = \mathit{Unary}(x_1)$, $v \rightarrow^{\beta *} x \wedge w \rightarrow^{\beta *} x$.
- Case $\underline{u = \mathit{Binary}(u_1, u_2)}$.
  - Case $\underline{v = \mathit{Binary}(u_2, u_1)}$. $v \rightarrow^{\beta_2} u$ and $u \rightarrow^{\beta_w} w$ where β_w corresponds to the specific reduction that lead to w. Hence there exists an $x = w$ because $v \rightarrow^{\beta_2} u \rightarrow^{\beta_w} x$.
  - Case $\underline{w = \mathit{Binary}(u_2, u_1)}$. Analogous to the case above, except that $x = v$.

- Case $\underline{u_1 \neq Empty}$ where $u_1 \rightarrow^{\beta} u_1'$ and $v = Binary(u_1', u_2)$. Then:
  - If $\underline{w = Binary(u_1'', u_2)}$ where $u_1 \rightarrow^{\beta} u_1''$.
    By IH $\exists u_1'''.(u_1' \rightarrow^{\beta *} u_1''' \ \wedge \ u_1'' \rightarrow^{\beta *} u_1''')$. Since $x = Binary(u_1''', u_2)$, done.
  - If $\underline{w = Binary(u_1, u_2')}$ where $u_2 \rightarrow^{\beta} u_2'$.
    Then $v \rightarrow^{\beta_w} Binary(u_1', u_2') \ \wedge \ w \rightarrow^{\beta_v} x$ where $x = Binary(u_1', u_2')$.
- Case $\underline{u_1 = Empty}$. From the assumption we know that $v \neq w$. Therefore we pick $v = u_2$ by (β_1). Then the only case not covered for w is for step $w = Binary(Empty, u_2')$ where $u_2 \rightarrow^{\beta_w} u_2'$.
  Then $v = u_2 \rightarrow^{\beta_w} u_2' = x$ and $w = Binary(Empty, u_2') \rightarrow^{\beta_1} u_2' = x$. Done.

q.e.d

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:

v	::=	λx : `T` . $nt \mid bv$	($values$)
bv	::=	`true` | `false`	($boolean\ values$)
nt	::=	$x \mid v \mid nt\ nt \mid$ `return` nt	($nested\ terms$)
tt	::=	$x \mid v \mid tt\ tt$	($top-level\ terms$)
t	::=	$nt \mid tt$	($terms$)
p	::=	tt	($programs$)
T	::=	`Bool` | `T` $\rightarrow$ `T`	($types$)

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 $\Gamma \vdash t$: `T`, you can use the $\Gamma \vdash t$: `T` \| `R`, where `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: λx : `Bool`. (`return 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`?

Solution:

Evaluation rules:

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

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

$$\frac{[x \to v_2]t_1 \longrightarrow^* \texttt{return}\ v_3}{(\lambda x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_1)\ v_2 \longrightarrow v_3} \quad \text{(E-AppAbs1)}$$

$$\frac{[x \to v_2]t_1 \longrightarrow^* v_3}{(\lambda x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_1)\ v_2 \longrightarrow v_3} \quad \text{(E-AppAbs2)}$$

$$\frac{t \longrightarrow t'}{\texttt{return}\ t \longrightarrow \texttt{return}\ t'} \quad \text{(E-Ret)}$$

$$(\texttt{return}\ v_1)\ t_2 \longrightarrow v_1 \quad \text{(E-AppRet1)}$$

$$t_1\ (\texttt{return}\ v_2) \longrightarrow v_2 \quad \text{(E-AppRet2)}$$

$$\texttt{return}\ (\texttt{return}\ v) \longrightarrow v \quad \text{(E-RetRet)}$$

Typing rules:

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

$$\frac{\Gamma,\ t\ \texttt{:}\ \texttt{T} \mid \texttt{R}}{\forall\ \texttt{A}.\ \Gamma \vdash \texttt{return}\ t\ \texttt{:}\ \texttt{A} \mid \{\texttt{T}\} \cup \texttt{R}} \quad \text{(T-Ret)}$$

$$\frac{\Gamma,\ x\ \texttt{:}\ \texttt{T}_1 \vdash t_2\ \texttt{:}\ \texttt{T}_2 \mid \emptyset}{\Gamma \vdash (\lambda x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_2)\ \texttt{:}\ \texttt{T}_1 \to \texttt{T}_2 \mid \emptyset} \quad \text{(T-Abs1)}$$

$$\frac{\Gamma,\ x\ \texttt{:}\ \texttt{T}_1 \vdash t_2\ \texttt{:}\ \texttt{T}_2 \mid \{\texttt{T}_2\}}{\Gamma \vdash (\lambda x\texttt{:}\ \texttt{T}_1\texttt{.}\ t_2)\ \texttt{:}\ \texttt{T}_1 \to \texttt{T}_2 \mid \emptyset} \quad \text{(T-Abs2)}$$

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

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

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

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.

Note: Remember to state clearly all the steps of your proof, including proofs of any lemmas that you use.

Solution:
We prove the Theorem by induction on the structure of `t`.

1. Let `t` be a variable `x`. The solution is immediate since `t` is closed and the case cannot occur.

2. Let `t` be an abstraction $\lambda x.t_1$. Since $\lambda x.t_1 \not\longrightarrow$, the solution is immediate.

3. Let `t` be an application $t_1\ t_2$. Then we can have three different cases, based on the used reduction rule for $t \longrightarrow t'$:

   (a) `E-APP1` - then $t_1 \longrightarrow t_1'$. As `t` is closed, then both t_1 and t_2 are closed as well. By induction t_1' is closed as well. Therefore $t_1' t_2$ is closed as well.

   (b) `E-APP2` - then $t_2 \longrightarrow t_2'$. As `t` is closed, then both v_1 and t_2 are closed as well. By induction t_2' is closed as well. Therefore $v_1 t_2'$ is closed as well.

   (c) `E-APPABS` - then $[x \to v_2]t_{12}$ for $(\lambda x.t_{12})v_2$. As `t` is closed, then both $\lambda x.t_{12}$ and v_2 are closed as well. In order to prove this, we need another lemma which says that substitution preserves the fact that terms are closed (stated and proven below). Done.

Lemma: For $\lambda x.t_1$ and t_2 that are closed, $[x \to t_2]t_1$ is closed as well.
Proof by induction on the structure of the lambda term t_1.

1. Let t_1 be a variable `y`. If $x = y$, then substitution results in t_2, which is closed.
   If $x \neq y$ then substitution results in t_1. Since from the assumption we know that t_1 is closed, therefore y was not free.

2. Let t_1 be an abstraction $\lambda y.t_1'$. If $x = y$, then the result is immediate, since x is bound in the abstraction, the t_1 is closed. If $x \neq y$, then by induction $[x \to t_2]t_1'$ is closed as well. Therefore $\lambda y.t_1'$ is closed.

3. Let t_1 be an application $t_1' t_2'$.
   Since t_1' and t_2' are closed, by induction $[x \to t_2]t_1'$ and $[x \to t_2]t_2'$ are closed as well. Therefore $[x \to t_2](t_1'\ t_2')$ is closed.

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.

t	::=	...	**terms**
	|	$\texttt{dynamic}\ t\ \texttt{as}\ T$	pack dynamic value
	|	$\texttt{typecase}\ t\ \texttt{match}\ \{\ \texttt{case}\ x_i^{i \in 1..n} : T_i \Rightarrow t_i\ \}$	unpack dynamic value
v	::=	...	**values**
	|	$\texttt{dynamic}\ v\ \texttt{as}\ \texttt{T}$	dynamic value
T	::=	...	**types**
	|	`Dynamic`	Dynamic type

Typing and evaluation rules:

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

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

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

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

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

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.

Solution:
Progress doesn't hold. One example is a missing case for a `Dynamic` type tag in a `typecase` construct: `typecase` (`dynamic` $\lambda x : Nat.0$ `as` $Nat \to Nat$) `match` {`case` $x : Nat \Rightarrow x$}
Preservation doesn't hold because we miss a necessary premise in `(T-Dynamic)` typing rule:
`typecase` (`dynamic` $\lambda x : Nat.0$ `as` Nat) `match case` $x : Nat \Rightarrow x$
and before the reduction step the type of the expression is `Nat` and afterwords it is `Nat` $\to$ `Nat`.

t	::=	...	**terms**
	|	`dynamic` t `as` T	pack dynamic value
	|	`typecase` t `match` { `case` $x_i^{i \in 1..n} : T_i \Rightarrow t_i$ `else` t}	unpack dynamic value
v	::=	...	**values**
	|	`dynamic` v `as` `T`	dynamic value
T	::=	...	**types**
	|	`Dynamic`	Dynamic type

Correct typing and evaluation rules:

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \mathtt{dynamic\ t\ as\ T} : \mathtt{Dynamic}} \quad \text{(T-Dynamic)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t} : \mathtt{Dynamic} \\ \Gamma, \mathtt{x}_i : \mathtt{T}_i \vdash \mathtt{t}_i : \mathtt{T} \\ \Gamma \vdash \mathtt{t}_{else} : \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{typecase\ t\ match\ \{case\ x}_i : \mathtt{T}_i \Rightarrow \mathtt{t}_i \ \mathtt{else\ t}_{else}\} : \mathtt{T}} \quad \text{(T-Typecase)}$$

$$\frac{\begin{array}{c}k \in 1..n \wedge \mathtt{T}_k = \mathtt{T} \\ for\ each\ j, j < k \ \Rightarrow\ \mathtt{T}_j \neq \mathtt{T}\end{array}}{\mathtt{typecase\ (dynamic\ v\ as\ T)\ match\ \{case\ x}_i^{i \in 1..n} : \mathtt{T}_i \Rightarrow \mathtt{t}_i \ \mathtt{else\ t}_{else}\} \to [\mathtt{x}_k \to \mathtt{v}]\,\mathtt{t}_k} \quad \text{(E-Typecase-Dynamic)}$$

$$\frac{for\ each\ j, j \in 1..n \ \Rightarrow\ \mathtt{T}_j \neq \mathtt{T}}{\mathtt{typecase\ (dynamic\ v\ as\ T)\ match\ \{\ case\ x}_i^{i \in 1..n} : \mathtt{T}_i \Rightarrow \mathtt{t}_i \ \mathtt{else\ t}_{else}\} \to \mathtt{t}_{else}} \quad \text{(E-Typecase-Dynamic-Else)}$$

$$\frac{\mathtt{t} \to \mathtt{t}'}{\mathtt{dynamic\ t\ as\ T} \to \mathtt{dynamic\ t}'\ \mathtt{as\ T}} \quad \text{(E-Dynamic)}$$

$$\frac{\mathtt{t} \to \mathtt{t}'}{\begin{array}{l}\mathtt{typecase\ t\ match\ \{\ case\ x}_i^{i \in 1..n} : \mathtt{T}_i \Rightarrow \mathtt{t}_i \ \mathtt{else\ t}_{else}\} \to \\ \mathtt{typecase\ t}'\ \mathtt{match\ \{\ case\ x}_i^{i \in 1..n} : \mathtt{T}_i \Rightarrow \mathtt{t}_i \ \mathtt{else\ t}_{else}\}\end{array}} \quad \text{(E-Typecase)}$$

Progress: Suppose t is a well-typed term (that is, t:T for some T). Then either t is a value or else there is some t' with $\texttt{t} \rightarrow \texttt{t'}$.
Missing cases:

- Case (T-Dynamic): $\texttt{t} = \texttt{dynamic t}' \texttt{ as T}, \texttt{t}' : \texttt{T}$
  By the induction hypothesis, either $\texttt{t}'$ is a value or else there is some $\texttt{t}''$ such that $\texttt{t}' \rightarrow \texttt{t}''$. If $\texttt{t}'$ is a value then the whole *packaged* dynamic value $\texttt{dynamic t}' \texttt{ as T}$ is a value. Otherwise rule E-Dynamic applies.

- Case (T-Typecase): $\texttt{t} = \texttt{typecase t}' \texttt{ match } \{ \texttt{case } \texttt{x}_i : \texttt{T}_i \Rightarrow \texttt{t}_i \texttt{ else } \texttt{t}_{else} \}$
  By the induction hypothesis, either $\texttt{t}'$ is a value or else there is some $\texttt{t}''$ such that $\texttt{t}' \rightarrow \texttt{t}''$. If $\texttt{t}'$ is not a value then rule (E-Typecase) applies. If $\texttt{t}'$ is a value and from premises we have that $\texttt{t}' : \texttt{T}'$ then by *canonical forms lemma* we know that $\texttt{t}'$ must be a dynamic value of a form $\texttt{dynamic t}'' \texttt{ as T}'$. Then either there exists a tag $\texttt{T}_i$ which matches $\texttt{T}'$ and rule (E-Typecase-Dynamic) applies, or there is none. For the latter case we have a fallback else case, and rule (E-Typecase-Dynamic-Else) applies.

Preservation (for completeness): If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{t} \rightarrow \texttt{t}'$, then $\Gamma \vdash \texttt{t}' : \texttt{T}$
Missing cases:

- Case (T-Dynamic): $\texttt{t} = \texttt{dynamic } \texttt{t}_1 \texttt{ as T}$
  From the premise of (T-Dynamic) we have that $\Gamma \vdash \texttt{t}_1 : \texttt{T}$. The only evaluation rule which applies is (E-Dynamic). And then $\texttt{t}'$ is of a form $\texttt{t}' = \texttt{dynamic } \texttt{t}_1' \texttt{ as T}$ and by induction hypothesis $\texttt{t}_1'$ is of type T. By (T-Dynamic) $\Gamma \vdash \texttt{dynamic } \texttt{t}_1' \texttt{ as T} : \texttt{T}$.

- Case (T-Typecase): $\texttt{t} = \texttt{typecase } \texttt{t}_1 \texttt{ match } \{ \texttt{case } \texttt{x}_i : \texttt{T}_i \Rightarrow \texttt{t}_i \texttt{ else } \texttt{t}_{else} \}$
  From the premise of (T-Typecase) we have that $\texttt{t}_1 : \texttt{Dynamic}$. Looking at the evaluation rules with typecase on the LHS we have three cases to consider:

  - Case (E-Typecase):
    A reduced term is of the form $\texttt{t}' = \texttt{typecase } \texttt{t}_1' \texttt{ match } \{ \texttt{case } \texttt{x}_i : \texttt{T}_i \Rightarrow \texttt{t}_i \texttt{ else } \texttt{t}_{else} \}$ and by induction hypothesis we know that $\texttt{t}_1' : \texttt{Dynamic}$. Since other premises (for *cases*) remain the same we have that $\texttt{t}' : \texttt{T}$ by (T-Typecase).
  - Case (E-Typecase-Dynamic):
    By the premise of the evaluation rule we have that $\texttt{t}_1 = \texttt{dynamic v as } \texttt{T}_v$ and the typing rule $\Gamma, \texttt{x}_k : \texttt{T}_k \vdash \texttt{t}_k : \texttt{T}$. The RHS of the evaluation rule gives us $\texttt{t}' = [x_k \rightarrow \texttt{v}]\,\texttt{t}_k$. By the premise of *k'th* case we know the type tag of the dynamic value $\texttt{T}_v = \texttt{T}_k$. Therefore by the substitution lemma we have that $\Gamma, \texttt{x}_k : \texttt{T}_v \vdash [x_k \rightarrow \texttt{v}]\,\texttt{t}_k : \texttt{T}$
  - Case (E-Typecase-Dynamic-Else):
    By the premise of the rule we know that none of the cases matched. Hence else case applies with $\Gamma \vdash \texttt{t}_{else} : \texttt{T}$ by the premise of (T-Typecase).

q.e.d

For reference: **predefined lambda terms**

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

$$
\begin{array}{ll}
\texttt{tru} = & \lambda t.\ \lambda f.\ t \\
\texttt{fls} = & \lambda t.\ \lambda f.\ f \\
\texttt{iszro} = & \lambda m.\ m\ (\lambda x.\ fls)\ tru \\
\texttt{test} = & \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f\ unit \\
 & \\
\texttt{pair} = & \lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s \\
\texttt{fst} = & \lambda p.\ p\ tru \\
\texttt{snd} = & \lambda p.\ p\ fls \\
 & \\
\texttt{c}_0 = & \lambda s.\ \lambda z.\ z \\
\texttt{c}_1 = & \lambda s.\ \lambda z.\ s\ z \\
\texttt{scc} = & \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z) \\
\texttt{plus} = & \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z) \\
\texttt{times} = & \lambda m.\ \lambda n.\ m\ (plus\ n)\ c_0 \\
 & \\
\texttt{zz} = & pair\ c_0\ c_0 \\
\texttt{ss} = & \lambda p.\ pair\ (snd\ p)\ (scc\ (snd\ p)) \\
\texttt{prd} = & \lambda m.\ fst\ (m\ ss\ zz) \\
 & \\
\texttt{fix} = & \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))
\end{array}
$$

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:

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

Evaluation rules:

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

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

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

Typing rules:

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

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

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

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

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