Operational Semantics of PCF

July 28, 2016

1 Definition of PCF

The grammar for types is:
$A ::= Nat\ |A \to B$

The grammar for expressions is:
$e ::=\ x \mid z \mid s(e) \mid case\ (e, z \to e_0, s(x) \to e_S) \mid e\ e' \ |\lambda x : A.e \mid fix\ x : A.\ e$

2 Type System for PCF

Γ is a context for types that maps variables to types. For an expression $x : A$ we have $\Gamma(x) = A$. The context of an expression e is given as $\Gamma \vdash e$.

We need a typing rule for each expression in PCF:

$$\overset{\text{VARIABLES}}{\frac{\Gamma(x) = A}{\Gamma \vdash x : A}} \qquad \overset{\text{ZERO}}{\frac{}{\Gamma \vdash z : Nat}} \qquad \overset{\text{SUCC}}{\frac{\Gamma \vdash e : Nat}{\Gamma \vdash s(e) : Nat}}$$

$$\overset{\text{CASE}}{\frac{\Gamma \vdash e : Nat \qquad \Gamma \vdash e_0 : A \qquad \Gamma, x : Nat \vdash e_s : A}{\Gamma \vdash\ case\ (e, z \to e_0, s(x) \to e_S)\ : A}}$$

$$\overset{\text{APPLICATION}}{\frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}} \qquad \overset{\text{ABSTRACTION}}{\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \to B}}$$

$$\overset{\text{FIX}}{\frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash\ fix\ x : A.\ e : A}}$$

3 Operational Semantics of PCF

We define the operational semantics of PCF using the *Call By Name* evaluation strategy, which means that function arguments are placed into the body of the function and evaluated within the entire function's evaluation, instead of before.

The semantics we define are *small step* semantics, which means that the transition relation $e \mapsto e'$ takes an expression e to another expression e' in only one step.

The first rules we have are congruence rules, which use the assumption $e \mapsto e'$ to replace e with e' in the whole expression:

$$\frac{e_0 \mapsto e_0'}{e_0\ e_1 \mapsto e_0'\ e_1} \qquad \frac{e \mapsto e'}{s(e) \mapsto s(e')}$$

$$\frac{e \mapsto e'}{case\ (e, z \to e_0, s(x) \to e_S) \mapsto case\ (e', z \to e_0, s(x) \to e_S)}$$

Then we define rules on individual expressions. Note that the case rule above is only defined on expressions that reduce. The one defined below is only defined for values (zero, successors of values, and lambda abstractions):

$$\frac{}{(\lambda x : A.\ e)\ e' \mapsto [e'/x]e}$$

$$\frac{}{case\ (z, z \to e_0, s(x) \to e_S) \mapsto e_0}$$

$$\frac{}{case\ (s(v), z \to e_0, s(x) \to e_S) \mapsto [v/x]e_S}$$

$$\frac{}{fix\ x : A.e \mapsto [fix\ x : A.e/x]e}$$

4 Lemmas for Type Safety

There are two Lemmas we must prove before proving Type Safety.

We also note that given a typing context Γ, an expression e and a type A such that we have the typing judgement $\Gamma \vdash e : A$, there is only one derivation of this judgement.

4.1 Weakening

Weakening is the following theorem, which says that for an expression of type A in a content Γ, adding another variable x to the context will not change the type of the expression:

Lemma 1. *If* $\Gamma \vdash e : A$ *then* $\Gamma, x : C \vdash e : A$

Proof. We prove this theorem by induction on derivation trees. If we have a derivation tree for our assumption then there exists a derivation tree for the conclusion. We can use the inductive hypothesis

$$\Gamma \vdash e : A \Rightarrow \Gamma, x : C \vdash e : A$$

As there is only one derivation for a given judgement $\Gamma \vdash e : A$, we can use induction on the possible expressions:

Variables We rename x to y using alpha equivalence. We assume $\Gamma \vdash y : A$, giving us the following derivation tree:

$$\frac{\Gamma(y) = A}{\Gamma \vdash y : A}$$

of which $\Gamma(y) = A$ is a subtree. Γ is a function, so is also a set of (variable, type) pairs. Therefore $\Gamma, x : C$ can be thought of as the set $\Gamma \cup \{(x, C)\}$. Therefore we define the function $(\Gamma, x : C)$, where $(\Gamma, x : C)(y) = A$ and for any other z $(\Gamma, x : C)(z) = \Gamma(z)$. Then we just use the typing rule for variables to get the following derivation tree:

$$\frac{(\Gamma, x : C)(y) = A}{\Gamma, x : C \vdash y : A}$$

Now we have a derivation tree for $\Gamma, x : C \vdash y : A$, so weakening holds for variables.

Zero We assume $\Gamma \vdash z : Nat$, giving us the following derivation tree:

$$\frac{}{\Gamma \vdash z : Nat}$$

. The typing rule for zero says that for any Γ, we have zero, because there are no assumptions. Therefore we can have $\Gamma, x : C$ as the context and get the following derivation tree:

$$\frac{}{\Gamma, x : C \vdash z : Nat}$$

Now we have a derivation tree for $\Gamma, x : C \vdash z : Nat$

z is a term of ground type, so can only have the type Nat, so we have proved this for all possible types of z.

Successor We assume $\Gamma \vdash s(e) : Nat$, giving us the following derivation tree from the typing rule:

$$\frac{\Gamma \vdash e : Nat}{\Gamma \vdash s(e) : Nat}$$

.

where $\Gamma \vdash e : Nat$ is a subtree. We can use the inductive hypothesis of weakening on this subtree to get $\Gamma, x : C \vdash e : Nat$. Then we use the typing rule for successor to get the following derivation tree:

$$\frac{\Gamma, x : C \vdash e : Nat}{\Gamma, x : C \vdash s(e) : Nat}$$

.

Now we have a derivation tree that gives us $\Gamma, x : C \vdash s(e) : Nat$, so weakening holds for the successor function, as it is only defined on terms of type Nat.

Case We assume $\Gamma \vdash\ \mathit{case}\ (e, z \rightarrow e_0, s(y) \rightarrow e_S)$, (renaming x to y using alpha equivalence) giving us the following derivation tree from the typing rule:

$$\frac{\Gamma \vdash e : Nat \qquad \Gamma \vdash e_0 : A \qquad \Gamma, y : Nat \vdash e_s : A}{\Gamma \vdash\ \mathit{case}\ (e, z \rightarrow e_0, s(y) \rightarrow e_S)\ : A}$$

giving us the subtrees $\Gamma \vdash e : Nat$, $\Gamma \vdash e_0 : A$ and $\Gamma, y : Nat \vdash e_s : A$ from the assumption of the typing rule. Using the inductive hypothesis on each of these, we get $\Gamma, x : C \vdash e : Nat$, $\Gamma, x : C \vdash e_0 : A$ and $\Gamma, y : Nat, x : C \vdash e_s : A$, so we can use the typing rule again with these assumptions:

$$\frac{\Gamma, x : C \vdash e : Nat \qquad \Gamma, x : C \vdash e_0 : A \qquad \Gamma, x : C, y : Nat \vdash e_s : A}{\Gamma, x : C \vdash\ \mathit{case}\ (e, z \rightarrow e_0, s(y) \rightarrow e_S)\ : A}$$

Now we have a derivation tree that gives us $\Gamma, x : C \vdash\ \mathit{case}\ (e, z \rightarrow e_0, s(y) \rightarrow e_S)\ : A$, so weakening holds for the case expression.

Application We assume $\Gamma \vdash e_0\ e_1$ giving us the following derivation tree:

$$\frac{\Gamma \vdash e_0 : A \to B \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e_0\ e_1 : B}$$

which gives us the subtrees $\Gamma \vdash e_0 : A \to B$ and $\Gamma \vdash e_1 : A$. Using the inductive hypothesis on these gives us $\Gamma, x : C \vdash e_0 : A \to B$ and $\Gamma, x : C \vdash e_1 : A$, so we can just use the typing rule for application again to get the following derivation tree:

$$\frac{\Gamma, x : C \vdash e_0 : A \to B \qquad \Gamma, x : C \vdash e_1 : A}{\Gamma, x : C \vdash e_0\ e_1 : B}$$

Therefore weakening holds for function application.

Abstraction We rename x to y using alpha equivalence. We assume $\Gamma \vdash \lambda y : A.\ e : A$, giving us the following derivation tree:

$$\frac{\Gamma, y : A \vdash e : B}{\Gamma \vdash \lambda y : A.\ e : A \to B}$$

which gives us the subtree $\Gamma, y : A \vdash e : B$. Using the inductive hypothesis, we get $\Gamma, y : A, x : C \vdash e : B$. Then we use the typing rule for λ abstraction to get the following tree:

$$\frac{\Gamma, y : A, x : C \vdash e : B}{\Gamma, x : C \vdash \lambda y : A.\ e : A \to B}$$

Now we have a derivation tree that gives us $\Gamma, x : C \vdash \lambda y : A.\ e : A \to B$, so weakening holds for λ abstraction.

Fixpoint We assume $\Gamma \vdash fix\, y : A.\ e : A$, renaming x to y using α equivalence. This gives us the following derivation tree:

$$\frac{\Gamma, y : A \vdash e : A}{\Gamma, y : A \vdash fix\ y : A.\ e : A}$$

As we have the subtree $\Gamma, y : A \vdash e : A$, we use the inductive hypothesis on this to get $\Gamma, x : C, y : A. \vdash e : A$. Then we use the typing rule for fix to get the following tree:

$$\frac{\Gamma, x : C, y : A \vdash e : A}{\Gamma, x : C, y : A \vdash fix\ y : A.\ e : A}$$

Therefore weakening holds for the fixpoint operator. Now we have proved weakening for derivation trees of any expression so weakening always holds.

□

4.2 Substitution Rules

Before we prove the Substitution Theorem, we must actually define the Substitution Rules for PCF, which are all instances of $[e/x]e'$. This says that an expression e replaces a variable x in another expression e'.

When defining the substitution rules, we must be careful that we avoid **variable capture** by bound variables. For example, given the following substitution:

$$[s(x)/y](\lambda x.x + y)$$

if we naively substitute $s(x)$ for y in $\lambda x.x + y$, we end up with $\lambda x + s(x)$ and the value of x is now the value assigned to the bound x. Therefore we can use **renaming** to avoid this.

Variables There are two cases for variables. The first is for when e' is the same as the variable being replaced:

$$[e/x]x = e$$

The second one is when e is a completely different variable, for which nothing happens:

$$[e/x]y = y$$

Zero For zero, any substitution will have no effect, as zero is a constant:

$$[e/x]z = z$$

Successor The successor function cannot be changed, so we substitute e in its argument:

$$[e/x]s(e') = s([e/x]e')$$

Case As we cannot change the case statement, we substitute e in the expressions given as arguments to the case statement, in the following way:

$$[e/x]\ (case\ (e', z \to e_0, s(x) \to e_s)) = case\ ([e/x]e', z \to [e/x]e_0, s(x) \to [e/x]e_s)$$

But we must be careful with variable capture, as the derivation of e_S is $\Gamma, x : Nat \vdash e_S$. If we have $[s(x)/y]e_S$ and $e_S = x + y$, then we have $x + s(x)$ and the value of $s(x)$ is bound by $\Gamma, x : Nat$. Therefore we should rename $s(x)$ in the case statement to something that is not free in e_S.

Therefore our rule for case will be:

$$(case\ (e', z \to e_0, s(x) \to e_s)) =$$

$$\begin{cases} (case\ ([e/x]e', z \to [e/x]e_0, s(x) \to [e/x]e_s)) & \text{if } x \notin FV(e') \\ (case\ ([e/x]e', z \to [e/x]e_0, s(y) \to [e/x]e_s)) & \text{if } x \in FV(e') \end{cases}$$

where $y \notin FV(e')$.

Application We substitute e in the function and its argument, then apply the new function to the new argument:

$$[e/x]e_0\ e_1 = [e/x]e_0\ ([e/x]e_1)$$

λ **Abstraction** There are two cases, the first when the bound variable is x. This does nothing, as we are just rewriting the function using alpha equivalence in this case:

$$[e/x](\lambda x : A.\ e') = \lambda x : A.\ e'$$

The second case is when the bound variable is not equal to x, where we substitute e in the expression. If y is in the free variables of e' then we must rename the bound variable to something else:

$$[e/x](\lambda y : A.e') =$$

$$\begin{cases} \lambda y : A.[e/x]e' & \text{if } y \notin FV(e') \\ \lambda z : A.[e/x]([z/y]e') & \text{if } y \in FV(e') \end{cases}$$

where $z \notin FV(e')$.

Fixpoint Fixpoint is similar to λ abstraction, so we have two cases, the first when the bound x is equal to x, for which we just rewrite the function using α equivalence. Therefore this rule does nothing:

$$[e/x]fix\ x : A.e' = fix\ e : A.e' =_\alpha fix\ x : A.e'$$

When the bound variable is not equal to x, we substitute e in the expression e'. If y is in the free variables of e' then we must rename the bound variable to something else:

$[e/x](fix\ y : A.e') =$

$$\begin{cases} fix\ y : A.[e/x]e' & \text{if } y \notin FV(e') \\ fix\ z : A.[e/x]([z/y]e') & \text{if } y \in FV(e') \end{cases}$$

where $z \notin FV(e')$.

4.3 Substitution

Now we have all the rules, we can prove Substitution, which is the following theorem. It says that if we have an expression e in the context Γ and an expression e' in the context Γ including a variable $x : A$, then the expression obtained by substituting e for x in e' will be derivable in the context Γ:

Lemma 2. *If* $\Gamma \vdash e : A$ *and* $\Gamma, x : A \vdash e' : C$ *then* $\Gamma \vdash [e/x]e' : C$

Proof. We can slso prove this by induction on derivation trees. We can rewrite the theorem as "If $D :: \Gamma \vdash e : A$ and $E :: \Gamma, x : A \vdash e' : C$ then $F :: \Gamma \vdash [e/x]e' : C$". We use induction on the tree E, as we are interested in the possible expressions e' could be. We can use the inductive hypothesis:

$$(\Gamma \vdash e : A \ \wedge \Gamma, x : A \vdash e' : C) \Rightarrow \ \Gamma \vdash [e/x]e' : C$$

As there is only one derivation for a given judgement $\Gamma \vdash e : A$, again we can use induction on the possible values of e':

Variables We assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash y : C$. As y is a different type to x, it cannot be equal to it, so there is only one case, as we can only use one of the substitution rules. The tree for y that is given is:

$$\frac{(\Gamma, x : A)(y) = C}{\Gamma, x : A \vdash y : C}$$

The function Γ is the set of pairs $(\Gamma, x : A)\backslash\{(x, A)\}$. As $x : A$ does not affect the value of y, we will still have $\Gamma(y) = C$. Using the type rule for variables, we get the tree:

$$\frac{\Gamma(y) = C}{\Gamma \vdash y : C}$$

$\Gamma \vdash y : C$ will be the same as $\Gamma \vdash [e/x]y : C$ using the substitution rule for variables, so we have the derivation tree needed.

Therefore variables satisfy substitution.

Zero We assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash z : Nat$. As z is a constant, it exists in any context Γ, so we have a tree for $\Gamma \vdash z : Nat$. This is equal to $\Gamma \vdash [e/x]z : Nat$, as this is always zero no matter what e and x are. Therefore as we already have the tree for $\Gamma \vdash z : Nat$, we use it as the derivation tree for $\Gamma \vdash [e/x]z : Nat$.

Successor We assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash s(e') : Nat$. The second tree is the following:

$$\frac{\Gamma, x : A \vdash e' : Nat}{\Gamma, x : A \vdash s(e') : Nat}$$

Therefore we have a subtree $\Gamma, x : A \vdash e' : Nat$. Using the induction hypothesis on this and $\Gamma \vdash e : A$, we have a derivation tree for $\Gamma \vdash [e/x]e' : Nat$. Using the typing rule for successor on this gives us the following tree:

$$\frac{\Gamma \vdash [e/x]e' : Nat}{\Gamma \vdash s([e/x]e') : Nat}$$

Using the substitution rule, we know the bottom half is equal to $[e/x]s(e')$, so we get the following derivation tree:

$$\frac{\Gamma \vdash [e/x]e' : Nat}{\Gamma \vdash [e/x]s(e') : Nat}$$

which is a derivation tree for $\Gamma \vdash [e/x]s(e') : Nat$. Therefore substitution holds for successor function.

Case We rename x to y using alpha equivalence, then assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash \mathit{case}\ (e', z \to e_0, s(y) \to e_S)\ : C$. The second derivation tree gives us the subtrees $\Gamma, x : A \vdash e' : \mathit{Nat}$, $\Gamma, x : A \vdash e_0 : C$ and $\Gamma, y : \mathit{Nat}, x : A \vdash e_s : C$. Using the induction hypothesis and the tree for $\Gamma \vdash e : A$, we get $\Gamma \vdash [e/x]e'$ and $\Gamma \vdash [e/x]e_0 : C$.

For $\Gamma, y : \mathit{Nat}, x : A \vdash e_s : C$, we need to change the context of e before we can apply the inductive hypothesis. We do this using weakening, which gives us $\Gamma, y : \mathit{Nat} \vdash e : A$. Now we apply the inductive hypothesis with this to get $\Gamma, y : \mathit{Nat} \vdash [e/x]e_s : C$.

Now we can apply the typing rule to these trees to get the following derivation tree:

$$\frac{\Gamma \vdash [e/x]e' : \mathit{Nat} \qquad \Gamma \vdash [e/x]e_0 : C \qquad \Gamma, y : \mathit{Nat} \vdash [e/x]e_s : C}{\Gamma \vdash\ \mathit{case}\ ([e/x]e', z \to [e/x]e_0, s(y) \to [e/x]e_S)\ : C}$$

By the substitution rule for case, we can replace the bottom half of the tree with $\Gamma \vdash [e/x]\ \mathit{case}\ (e', z \to e_0, s(y) \to e_S)\ : C$. Therefore substitution holds for the case statement.

Application We assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash e_0\ e_1 : B$. The second tree is the following:

$$\frac{\Gamma, x : A \vdash e_0 : A \to B \qquad \Gamma, x : A \vdash e_1 : A}{\Gamma, x : A \vdash e_0\ e_1 : B}$$

which contains the subtrees $\Gamma, x : A \vdash e_0 : A \to B$ and $\Gamma, x : A \vdash e_1 : A$. Combining each of these trees with $\Gamma \vdash e : A$ and the inductive hypothesis gives us $\Gamma \vdash [e/x]e_0 : A \to B$ and $\Gamma \vdash [e/x]e_1 : A$. Using the typing rule for function application with these trees gives us a derivation tree for $\Gamma \vdash [e/x]e_0([e/x]e_1) : C$. This is equal to $\Gamma \vdash [e/x]e_0\ e_1 : C$, so we have the derivation tree for this judgement.

Therefore substitution holds for function application.

Abstraction We rename x to y using α equivalence. Then derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash \lambda y : A.\ e' : A \to B$. For the second tree we have the following:

$$\frac{\Gamma, x : A, y : A \vdash e' : B}{\Gamma, x : A \vdash \lambda y : A.\ e' : A \to B}$$

which gives us the subtree $\Gamma, x : A, y : A \vdash e' : B$. Then we use weakening on $\Gamma \vdash e : A$ to get $\Gamma, y : A \vdash e : A$, which when used with the inductive hypothesis gives us $\Gamma, y : A \vdash [e/x]e' : B$.

Applying the typing rule for λ abstraction to this gives us the following tree:

$$\frac{\Gamma, y : A \vdash [e/x]e' : B}{\Gamma \vdash \lambda y : A.\ [e/x]e' : A \to B}$$

.

Now there are two cases:

1. When $y \notin FV(e')$, the substitution rule gives us $\lambda y : A.\ [e/x]e' = [e/x]\lambda y : A.\ e'$, so we have a derivation tree for $\Gamma \vdash [e/x]\lambda y : A.\ e' : B$
2. When $y \in FV(e')$, rewrite $\lambda y : A.\ [e/x]e' : A \to B$ as $\lambda z : A.[e/x]([z/y]e')$. Then this is the same as $[e/x]\lambda y : A.\ e'$, so we have a derivation tree for $\Gamma \vdash [e/x]\lambda y : A.\ e' : B$

Fixpoint We rename x to y using alpha equivalence, then assume derivation trees exist for $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash fix\ y : C.\ e' : C$. The second tree is the following:

$$\frac{\Gamma, y : C, x : A \vdash e' : C}{\Gamma, x : A \vdash fix\ y : C.e' : C}$$

giving us the subtree $\Gamma, y : C, x : A \vdash e' : C$. Then we use weakening on $\Gamma \vdash e : A$ to get $\Gamma, y : C \vdash e : A$, which when used with the inductive hypothesis gives us $\Gamma, y : C \vdash [e/x]e' : C$.

Applying the typing rule for fixpoint to this gives us the following derivation tree:

$$\frac{\Gamma, y : C \vdash [e/x]e' : C}{\Gamma \vdash fix\ y : C.[e/x]e' : C}$$

Now there are two cases:

1. When $y \notin FV(e')$, the substitution rule for fixpoint gives us $\Gamma \vdash fix\ y : C.[e/x]e' : C = \Gamma \vdash [e/x](fix\ y : C.e') : C$, so we have the required derivation tree and substitution holds for fixpoint.
2. When $y \in FV(e')$, rewrite $fix\ y : C.\ [e/x]e' : C$ as $fix\ z : C.[e/x]([z/y]e')$. Then this is the same as $[e/x]fix\ y : C.\ e'$, so we have a derivation tree for $\Gamma \vdash [e/x]fix\ y : C.\ e' : C$

Now we have proved substitution holds for derivation trees of any expression e'. □

5 Type Safety Lemmas

There are two type safety lemmas we must prove:

5.1 Type Preservation

Type preservation says that if an expression e has type A in a context Γ, and it evaluates in one step to e', then e' must also have type A in Γ:

Lemma 3. *If $\Gamma \vdash e : A$ and $e \mapsto e'$, then $\Gamma \vdash e' : A$*

Proof. We can prove this by induction on derivation trees, so we rewrite the theorem as "If $D :: \Gamma \vdash e : A$ and $E :: e \mapsto e'$ (in one step) then $\exists.F :: \Gamma \vdash e' : A$:

There will be a derivation tree E for each rule in the operational semantics, where D is the tree for the typing derivation of the first half of the rule represented by E.

Therefore we can check this statement for every evaluation rule on every possible expression:

Vsriables There are no rules in the operational semantics for when e is just a variable so we do nothing here.

Zero Same as for variables.

Successor We have the following tree for D , given by the typing rule of successor:

$$\frac{\Gamma \vdash e : Nat}{\Gamma \vdash s(e) : Nat}$$

The tree for E will be the congruence rule for successor:

$$\frac{e \mapsto e'}{s(e) \mapsto s(e')}$$

From these two trees, we get the subtrees $\Gamma \vdash e : Nat$ and $e \mapsto e'$. Using the inductive hypothesis of type preservation we get a tree for $\Gamma \vdash e' : Nat$. Then using the typing rule for successor with this we get a tree for $\Gamma \vdash s(e') : Nat$.

There are no other rules when the expression is $s(e)$, so type preservation holds for successor expressions.

Case There are three evaluation rules, which depend on the expression e being checked:

1. If e can be reduced, then we use the typing rule for case as D:

   $$\frac{\Gamma \vdash e : Nat \qquad \Gamma \vdash e_0 : A \qquad \Gamma, x : Nat \vdash e_s : A}{\Gamma \vdash \ \textit{case}\ (e, z \to e_0, s(x) \to e_S)\ : A}$$

   and congruence evaluation rule for case as E:

   $$\frac{e \mapsto e'}{\textit{case}\ (e, z \to e_0, s(x) \to e_S) \mapsto \textit{case}\ (e', z \to e_0, s(x) \to e_S)}$$

   Then we have subtrees for $\Gamma \vdash e : Nat$, $\Gamma \vdash e_0 : A$, $\Gamma, x : Nat \vdash e_s : A$ and $e \mapsto e'$.

   Using the inductive hypothesis of type preservation, with the trees for $\Gamma \vdash e : Nat$ and $e \mapsto e'$ we get a tree for $\Gamma \vdash e' : Nat$. Then we apply the typing rule for case with this, $\Gamma \vdash e_0 : A$ and $\Gamma, x : Nat \vdash e_s : A$ to get a tree for $\Gamma \vdash \textit{case}\ (e', z \to e_0, s(x) \to e_S) : A$

2. If $e = \textit{case}\ (z, z \to e_0, s(x) \to e_S)$, then D is

   $$\frac{\Gamma \vdash z : Nat \qquad \Gamma \vdash e_0 : A \qquad \Gamma, x : Nat \vdash e_s : A}{\Gamma \vdash \ \textit{case}\ (z, z \to e_0, s(x) \to e_S)\ : A}$$

   and E is the tree for the evaluation rule $\textit{case}\ (z, z \to e_0, s(x) \to e_S) \mapsto e_0$. $\Gamma \vdash e_0 : A$ is a subtree of D, so we already have the tree for $\Gamma \vdash e_0 : A$

3. If $e = \textit{case}\ (s(v), z \to e_0, s(x) \to e_S)$, then D is

   $$\frac{\dfrac{\Gamma \vdash v : Nat}{\Gamma \vdash s(v) : Nat} \qquad \Gamma \vdash e_0 : A \qquad \Gamma, x : Nat \vdash e_s : A}{\Gamma \vdash \ \textit{case}\ (z, z \to e_0, s(x) \to e_S)\ : A}$$

   and E is the tree for the evaluation rule $\textit{case}\ (z, s(v), \to e_0, s(x) \to e_S) \mapsto [v/x]e_s$.

   We have the subtrees for $\Gamma \vdash v : Nat$, $\Gamma \vdash s(v) : Nat$, $\Gamma \vdash e_0 : A$, $\Gamma, x : Nat \vdash e_s : Nat$ from the tree D.

   We get the tree for $\Gamma \vdash [v/x]e_s : A$ by using the substitution lemma, with the subtrees for $\Gamma \vdash v : Nat$ and $\Gamma, x : Nat \vdash e_s : A$ as parameters.

Application There are two evaluation rules for function application:

1. When e is a function that can be reduced further, we use the congruence evaluation rule for application, which gives us the following tree for D:

$$\frac{\Gamma \vdash e_0 : A \to B \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash e_0\ e_1 : B}$$

and the following tree for E:

$$\frac{e_0 \mapsto e_0'}{e_0\ e_1 \mapsto e_0'\ e_1}$$

We use the induction hypothesis with the subtrees for $\Gamma \vdash e_0 : A \to B$ and $e_0 \mapsto e_0'$ to get a subtree for $\Gamma \vdash e_0' : A \to B$. Then using this and the subtree for $\Gamma \vdash e_1 : B$, in the typing rule for function application, we get a subtree for $\Gamma \vdash e_0'\ e_1 : B$.

2. When $e = (\lambda x : A.\ e)\ e'$, we have the following tree for D:

$$\frac{\dfrac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A).e : A \to B} \qquad \Gamma \vdash e' : A}{\Gamma \vdash (\lambda x : A).e\ e' : B}$$

And the tree for $(\lambda x : A).e\ e' \mapsto [e'/x]e$ for E. We have subtrees $\Gamma\ \vdash e' : A$ and $\Gamma, x : A \vdash e : B$, so we use these as parameters to the Substitution Lemma to get the tree for $\Gamma \vdash [e'/x]e : B$.

λ-abstraction has no evaluation rules when taken as a single expression

Fixpoint When $e = fix\ x : A.e$, we have the following tree for D:

$$\frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash\ fix\ x : A.\ e : A}$$

and E is the tree for the rule $fix\ x : A.\ e \mapsto [fix\ x : A.e/x]e$. We already have the tree for $\Gamma \vdash\ fix\ x : A.\ e : A$ as an assumption and $\Gamma, x : A \vdash e : A$ is a subtree of it, so we can use the substitution lemma with these parameters to get a tree for $\Gamma \vdash [fix\ x : A.e/x]e : A$

Now we have proved type preservation for all the rules in the operational semantics on all possible expressions. □

5.2 Type Progress

Type progress says that if an expression e has type A in a context Γ, then it must evaluate to another expression e' in one step, or be a value (so e cannot be evaluated further), where possible values are

$$v :: z \mid s(v) \mid \lambda x : A.\ e$$

which are numbers or non-recursive functions.(So note this is not saying it terminates, or that it has normal form.):

Lemma 4. *If* $\vdash e : A$ *then* $e \mapsto e'$ *or* e *is a value.*

Proof. We can prove this by induction on derivation trees of e, so we rewrite the theorem as "If $D :: \vdash e : A$ then $E :: e \mapsto e'$ or e is a value".

When we have a derivation tree D for a closed term e, we either get a derivation tree E, for evaluating it in one step to another expression, or e is a value.

Zero z is a value

Variables There are no closed terms that are variables, so this is vacuously true. This is because the context of a term is a set of $(variable, type)$ pairs, so when it is empty, there are no variables. Therefore we have no derivation tree for $\vdash x : A$, where x is a variable, so there is no case for variables.

Successor When we have an expression $s(e)$, we assume $\vdash s(e) : Nat$, giving us the following tree:

$$\frac{\vdash e : Nat}{\vdash s(e) : Nat}$$

so we have a subtree for $\vdash e : Nat$. We can use induction on this subtree to get $e \mapsto e'$ or e is a value. We can assume either side of this:

1. When $e \mapsto e'$ we use the congruence rule for successor to get a tree for $s(e) \mapsto s(e')$. Therefore $s(e) \mapsto s(e')$ or $s(e)$ is a value
2. When e is a value, v, we rewrite $s(e)$ to $s(v)$ This is a value, so $s(e) \mapsto s(e')$ or $s(e)$ is a value is true

Case When we have an expression $\mathit{case}\ (e, z \mapsto e_0, s(x) \mapsto e_S)$, we assume $\vdash \mathit{case}\ (e, z \mapsto e_0, s(x) \mapsto e_S) : A$, giving us the following tree:

$$\frac{\vdash e : \mathit{Nat} \qquad \vdash e_0 : A \qquad x : \mathit{Nat} \vdash e_s : A}{\vdash\ \mathit{case}\ (e, z \rightarrow e_0, s(x) \rightarrow e_S)\ \ : A}$$

,so we have subtrees for $\vdash e : \mathit{Nat}$, $\vdash e_0 : A$ and $x : \mathit{Nat} \vdash e_s : A$.

By induction on $\vdash e : \mathit{Nat}$, we know that $e \mapsto e'$ or e is a value. We can assume either side of this:

1. When $e \mapsto e'$, we use the congruence rule for case to get $\mathit{case}\ (e', z \rightarrow e_0, s(x) \rightarrow e_S)\ \ : A$. Therefore our original expression maps to some e', so we have $\mathit{case}\ (e, z \mapsto e_0, s(x) \mapsto e_s) \mapsto e'$ for some e'

2. When e is a value there are two cases.:

   (a) $e = z$. The evaluation rule for this has no assumption, so we already have a tree that maps $\mathit{case}\ (z, z \rightarrow e_0, s(x) \rightarrow e_S)$ to another expression, which is e_0.

   (b) $e = s(v)$. The evaluation rule for this also has no assumption, so we already have a tree that maps $\mathit{case}\ (s(v), z \rightarrow e_0, s(x) \rightarrow e_S)$ to another expression, which is $[v/x]e_s$

Therefore, no matter what e is in the case expression, it always maps to another expression, e', so ”‘$\mathit{case}\ (e, z \rightarrow e_0, s(x) \rightarrow e_S) \mapsto e'$ or $\mathit{case}\ (e, z \rightarrow e_0, s(x) \rightarrow e_S)$ is a value”’ is true.

Application When we have an expression $e_0\ e_1$, we assume $\vdash e_0\ e_1 : B$ giving us the following tree:

$$\frac{\vdash e_0 : A \rightarrow B \qquad \vdash e_1 : A}{\vdash e_0\ e_1 : B}$$

so we have a subtree for $\vdash e_0 : A \rightarrow B$. We can use the inductive hypothesis on this to get $e_0 \mapsto e_0$’ or e_0 is a value. We can assume either side of this:

1. When $e_0 \mapsto e_0'$, we use the congruence rule for application to get a tree for $e_0\ e_1 \mapsto e_0'\ e_1$

2. When e_0 is a value it must be $\lambda x : A.e_0$, as the other values are not function types. The evaluation rule for $(\lambda x : A.e_0)\ e_1$ has no assumption, so we already have a tree that maps $(\lambda x : A.e_0)\ e_1$ to another expression, which is $[e_1/x]e_0$

Therefore, for every possible value of $e_0\ e_1$, we can evaluate it to another expression in one step, so ”$e_0\ e_1 \mapsto e_0'\ e_1$ or $e_0\ e_1$ is a value” is true.

λ-abstraction $\lambda x : A.e$ is always a value, for any $x : A$ ane $e : A$

Fixpoint When we have an expression $fix\ x : A.\ e$, we assume $\vdash\ fix\ x : A.\ e : A$, giving us the following tree:

$$\frac{x : A \vdash e : A}{\vdash\ fix\ x : A.\ e : A}$$

There are no assumptions in the evaluation rule for fixpoint, so we already have the tree that maps $fix\ x : A.\ e$ to another expression, which is $[fix\ x : A.e/x]e$. Therefore we know that $fix\ x : A.\ e$ always maps to another expression, so "$fix\ x : A.\ e$ maps to another expression e' in one step, or $fix\ x : A.\ e$ is a value" is true.

Now we have proved type progress for all possible expressions. □

6 Type Safety

Type Safety can be proved as a corollary of the two lemmas we have just proved. Usually this is just assumed because of the two lemmas.

The theorem for type safety says that an closed term either evaluates to a value in a finite number of steps or loops forever:

Lemma 5. *If $\vdash e : A$ then $e \mapsto^* v$ (where $\vdash v : A$) or $e \mapsto^\infty$*

Proof. We can prove this by coinduction, because the evaluation is infinite.

□