Lecture Notes 10
Observational Equivalence

Carlo Angiuli

B522: PL Foundations
April 7, 2025

We now return to the question of when two programs, or more generally two terms, should be considered equal. We have already seen that the relation of *evaluating to identical values* is rather too strict. For one thing, it fails to equate functions that produce the same output on every input (which we called *extensionally equal* functions in the lecture on type isomorphism). In addition, this relation does not give us any insight into the equality of *open* terms.

In this lecture we will develop the basic theory of *observational equivalence*, also known as (Morris-style [Mor69]) *contextual equivalence* or *extensional equivalence*, one of the standard notions of program fragment equivalence considered in programming language theory. We will then discuss the pros and cons of observational equivalence, and introduce a more tractable characterization of observational equivalence known as *logical equivalence*.

Observational equivalence for System T and PCF are discussed in Chapters 46 and 47 of Harper [Har16] respectively.

1 Desiderata

Let's take a step back and think about the properties that equality should satisfy.

- It should be a binary relation on two terms of the same type.
- It should be an *equivalence relation*: reflexive (every term should be equal to itself), symmetric (if $e = e'$ then $e' = e$), and transitive (if $e = e'$ and $e' = e''$ then $e = e''$).
- It should also be a *congruence*, i.e., it should be preserved by all term formers. For example, if $e = e'$ then we should also have $\mathtt{fst}(e) = \mathtt{fst}(e')$, $f\ e = f\ e'$, $\lambda x : \tau.e = \lambda x : \tau.e'$, etc.

These are the basic properties of any equality relation; they allow us to chain together equalities and "replace equals by equals" anywhere inside a larger term anywhere inside a larger term. In particular, transitivity and congruence combine to give us more general equational reasoning: if $f = f'$ and $e = e'$ then we have $f\ e = f'\ e'$ by $f\ e = f\ e' = f'\ e'$.

The last example of congruence, that λ should preserve equality, implies that

- Equality should be defined not only for closed terms but also *open terms*.

That is, for every Γ, every τ, and every pair of terms $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$, we should have an equality relation $\Gamma \vdash e = e' : \tau$, and if (for example) $\Gamma, x : \tau_1 \vdash e = e' : \tau_2$ then $\Gamma \vdash \lambda x : \tau_1.e = \lambda x : \tau_1.e' : \tau_1 \to \tau_2$.

Remark 10.1. In some sense one really has a separate equivalence relation on terms for every Γ and τ, although congruence is a very strong coherence condition between these relations.

So far our considerations have been very syntactic, referencing the type system but not the operational semantics. Taking evaluation into account,

- Equality of closed terms should contain evaluation: if $\cdot \vdash e : \tau$ and $e \longmapsto e'$, then $\cdot \vdash e = e' : \tau$.
- Equality (of both open terms and closed terms) should be somehow defined in terms of the operational semantics, and evaluation should not be able to distinguish equal terms.

The latter point is crucial because the point of writing programs is to run them, and the point of reasoning about programs is to better understand what will happen when you run them. Just as the type system gives us some guarantees about the runtime behavior of programs, we would like equality to give us some fairly strong guarantees about the "sameness" of the results obtained from running each program. It is also somewhat tricky, though, because evaluation is only defined for closed terms and equality must be defined for open terms as well.

These final considerations are less mandatory from a theoretical perspective but are generally desirable for other reasons:

- At function types, it should contain extensional equality. (Otherwise it will not be very useful).
- It should generalize all of the principal reductions to open terms (i.e., it should contain β-equivalence). For example, if $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash e_2 : \tau_2$ then $\Gamma \vdash \texttt{fst}((e_1, e_2)) = e_1 : \tau_1$. (This lets us perform various simplifications.)
- It should be "canonical"; the "best" equality relation in some sense.

2 Observations

How can we transform evaluation from a deterministic relation on closed terms to a congruence relation on open terms? Given our experience with closing substitutions in the definition of hereditary termination, we might imagine saying that two open terms are equal if for all well-typed terms that we substitute for their variables, they evaluate to the same value. There are a few problems with this definition, but perhaps the easiest one to point out is that it does not equate extensionally equal functions such as $\lambda x : \tau_1 \times \tau_2 . x$ and $\lambda x : \tau_1 \times \tau_2 . (\texttt{fst}(x), \texttt{snd}(x))$.

In fact, the entire fact that functions evaluate to λ terms is a technical simplification of structural operational semantics. In Racket, the REPL reports the value of functions as `#<procedure>`, not `(lambda ...)`; even the interpreters in C311/B521 evaluate functions to closures. So our first step toward extensional equality is to ignore that we can "look under the hood" of function values.

We thus draw a distinction between program outcomes that are *observable* and those that are not. Observable values include "data" such as `unit`, booleans, natural numbers, or lists of data. In PCF, it is also observable whether or not a program terminates. For each of these observations it should be quite clear-cut—regardless of how we implement our language—whether two observations are the same or different. (And all implementations should agree!) On the other hand, λs are the prototypical example of *non-observable* values.

We will define *observational equivalence* as follows. First, we pick a type and a notion of observation for (closed) programs of that type. For example, we might say that `bool` is the *observable* type, with the two possible observations of a program $\cdot \vdash e : \texttt{bool}$ being evaluating to `true` and evaluating to `false`. In that case, we only permit ourselves to observe the behavior of closed terms of type `bool`.

For terms of any other type or in any other typing context, we consider all possible surrounding programs into which that term could fit. For example, given the term $x : \texttt{nat} \vdash \texttt{plus}\ x : \texttt{nat} \to \texttt{nat}$ we can imagine many ways of "completing" it into a program of type `bool`:

$$\texttt{zero?}\ ((\lambda x : \texttt{nat}.\texttt{plus}\ x\ x)\ \texttt{zero})$$
$$\texttt{zero?}\ (\texttt{case}_{\texttt{nat}}\ \texttt{suc(zero)}\ [\texttt{zero} \to \texttt{zero}]\ [\texttt{suc}(x) \to \texttt{plus}\ x\ \texttt{suc(zero)}])$$
$$(\lambda f : \texttt{nat} \to \texttt{nat}.\texttt{true})\ (\lambda x : \texttt{nat}.\texttt{plus}\ x)$$
$$\vdots$$

Note that some of these evaluate to `true`, and others to `false`; some of these "use" `plus` x in an essential way, whereas in others we never evaluate `plus` at all.

For any pair of terms $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$, we say that they are observationally equivalent if *there is no surrounding context that distinguishes them*, that

is, if there is no single way to "complete" them into programs of type `bool` that causes e to be completed into a program computing `true` and e' into a program computing `false`. If we think of each of these contexts as an "experiment" that we can run on e or e', then observationally equivalent terms are those that no experiment can distinguish.

We will make this more precise momentarily, but let us first discuss choosing a notion of observation. Besides the programming language itself, the notion of observation is the only "input" to the definition of observational equivalence, so one might think that it must be chosen very carefully.

In fact, most choices turn out to produce identical results. Suppose that we choose `nat` as our observable type instead of `bool`. If two programs compute different natural numbers, say `zero` and `suc(zero)`, then it is easy to surround them in a context that will cause them to compute two different booleans (e.g., testing whether they are `zero?`). On the other hand, if there is no surrounding context that causes e and e' to produce different numbers, then there also cannot be any surrounding context in which they produce different booleans: for if there were, it would give us a way to distinguish e and e' numerically as well (e.g., `if(`$-$`, zero, suc(zero))`).

In languages with nontermination, such as PCF, it is common to choose termination (at any type, such as `unit`) as the observation. For one, it is necessary in any case to include nontermination as one of the possibilities: two programs of type `bool` produce the same result if both evaluate to `true`, both to `false`, or both diverge. And as above, terms that are (in)distinguishable with respect to one of these observations are also (in)distinguishable with respect to the other.

Remark 10.2. The foregoing discussion hints at one downside to observational equivalence: it is very sensitive to what features are in one's language. Many languages have an extremely fine observational equivalence because of the ability to compare functions for equality, measure how long a computation takes, etc. In the other direction, if we forget to include `if` in our language, then `true` and `false` may inadvertently become observationally equivalent with respect to `nat` observations.

We will now carefully define observational equivalence for the STLC, or at least a variation on it. We cannot use the STLC itself because it does not have any suitable notion of observation: the only data is `unit`, and there is only one observable outcome at `unit`, namely terminating with value `()`! (If these are our observations, then all terms are observationally equivalent.)

We therefore make the simplest possible change to the STLC, which is to replace `unit` by a type that has two different values. We call it the type of "`answers`" and call its two values `yes` and `no`. The rules are as follows:

$$
\begin{array}{llllll}
\textit{Types} & \tau ::= & \vdots & \vdots & \vdots & \\
 & & \mathsf{ans} & \mathsf{ans} & \text{answer type} & \\
\textit{Terms} & e ::= & \vdots & \vdots & \vdots & \\
 & & \mathsf{yes} & \mathsf{yes} & \text{yes answer} & \\
 & & \mathsf{no} & \mathsf{no} & \text{no answer} &
\end{array}
$$

$$
\ldots \qquad \frac{}{\mathsf{yes\ val}} \qquad\qquad \frac{}{\mathsf{no\ val}}
$$

$$
\ldots \qquad \frac{}{\Gamma \vdash \mathsf{yes} : \mathsf{ans}} \qquad\qquad \frac{}{\Gamma \vdash \mathsf{no} : \mathsf{ans}}
$$

Remark 10.3. An alternative would be to add booleans to the STLC, but those are more complicated because of `if`. Although `ans` is rather useless for programming—we cannot actually *use* its terms in any way—it is the simplest addition that gives us the ability to distinguish, for example, $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$.

3 Observational equivalence for STLC++

Definition 10.4. A *term context* (or *expression context*) C, not to be confused with a typing context Γ, is a term with one hole $\circ$ in it. We define term contexts inductively as follows:

$$
\begin{array}{llll}
\textit{Term contexts} & C ::= & \circ \mid \lambda x : \tau.C \mid C\ e \mid e\ C & \\
 & \mid & (C, e) \mid (e, C) \mid \mathtt{fst}(C) \mid \mathtt{snd}(C) &
\end{array}
$$

An example of a term context is $\mathtt{fst}(\mathsf{yes}, (\lambda x : \mathsf{ans}.(x, \circ)))$. Term contexts can be seen as a generalization of evaluation contexts where the hole can be anywhere whatsoever, including under binders.

Definition 10.5. For any term context C and term e, we define the instantiation

of C with e, written $C\{e\}$, by structural recursion:

$$\begin{aligned}
\circ\{e\} &\coloneqq e \\
(\lambda x : \tau.C)\{e\} &\coloneqq \lambda x : \tau.C\{e\} \\
(C\ e')\{e\} &\coloneqq C\{e\}\ e' \\
(e'\ C)\{e\} &\coloneqq e'\ C\{e\} \\
(C, e')\{e\} &\coloneqq (C\{e\}, e') \\
(e', C)\{e\} &\coloneqq (e', C\{e\}) \\
\mathtt{fst}(C)\{e\} &\coloneqq \mathtt{fst}(C\{e\}) \\
\mathtt{snd}(C)\{e\} &\coloneqq \mathtt{snd}(C\{e\})
\end{aligned}$$

Note that term context instantiation *captures* variables in e. For example, if $C = \lambda x : \tau.\circ$ and $e = x$, then instantiating $C\{e\} = \lambda x : \tau.x$ changes x from a free variable to a bound variable. For this reason, context instantiation is not a special case of substitution; we cannot treat term contexts as terms with a distinguished "hole" variable and instantiation as substitution for that variable.

Variable capture is intentional for this definition, because it accurately expresses the idea that we want to consider "any surrounding context," which in the case of variables includes "any binding site."

Definition 10.6 (Term context typing)**.** For typing contexts Γ, Γ' and types τ, τ' we define the judgment $C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ inductively as follows:

$$\frac{}{\circ : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma \rhd \tau)} \qquad \frac{C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma', x : \tau_1 \rhd \tau_2)}{\lambda x : \tau_1.C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \to \tau_2)}$$

$$\frac{C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \to \tau_2) \qquad \Gamma' \vdash e_1 : \tau_1}{C\ e_1 : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2)}$$

$$\frac{C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1) \qquad \Gamma' \vdash f : \tau_1 \to \tau_2}{f\ C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2)}$$

$$\frac{C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1) \qquad \Gamma' \vdash e_2 : \tau_2}{(C, e_2) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \times \tau_2)}$$

$$\frac{C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2) \qquad \Gamma' \vdash e_1 : \tau_1}{(e_1, C) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \times \tau_2)}$$

$$\frac{C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \times \tau_2)}{\mathtt{fst}(C) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1)} \qquad \frac{C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_1 \times \tau_2)}{\mathtt{snd}(C) : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau_2)}$$

Lemma 10.7. *If* $C : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ *and* $\Gamma \vdash e : \tau$ *then* $\Gamma' \vdash C\{e\} : \tau'$.

Given two closed programs of observable type, the relation of "having the same observable outcome" is often known as Kleene equivalence.

Definition 10.8 (Kleene equivalence)**.** Given two programs $\cdot \vdash e : \mathtt{ans}$ and $\cdot \vdash e' : \mathtt{ans}$, we say that e and e' are *Kleene equivalent*, written $e \simeq e'$, if either $e \Downarrow \mathtt{yes}$ and $e' \Downarrow \mathtt{yes}$, or $e \Downarrow \mathtt{no}$ and $e' \Downarrow \mathtt{no}$.

Kleene equivalence only applies to closed programs of type ans so it is not a congruence and lacks various other desirable properties, but it is reflexive, symmetric, transitive, and closed under head expansion and head reduction.

Definition 10.9 (Observational equivalence)**.** Given two terms $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$, we say that e and e' are *observationally equivalent*, $\Gamma \vdash e \cong e' : \tau$, if for every $C : (\Gamma \rhd \tau) \rightsquigarrow (\cdot \rhd \mathtt{ans})$ we have $C\{e\} \simeq C\{e'\}$.

Observational equivalence is also reflexive, symmetric, and transitive, but unlike Kleene equivalence, it is defined for every Γ and τ.

Lemma 10.10. *Observational equivalence is a* congruence*: that is, it is an equivalence relation, and if* $\Gamma \vdash e \cong e' : \tau$ *and* $\mathcal{D} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$*, then* $\Gamma' \vdash \mathcal{D}\{e\} \cong \mathcal{D}\{e'\} : \tau'$.

Lemma 10.11. *Observational equivalence is* consistent *with Kleene equivalence: if* $\cdot \vdash e \cong e' : \mathtt{ans}$ *then* $e \simeq e'$.

computational adequacy

Lemma 10.11 implies in particular that observational equivalence does not equate all terms; if it did then it would equate yes and no, but these are clearly not Kleene equivalent.

All proper notions of equality should be consistent congruences: they should be preserved by every term former and should generalize Kleene equivalence. It turns out that observational equivalence is the *coarsest* notion of equality—the one that equates the most terms—in the sense that whenever there is a consistent congruence that relates e and e', then e and e' are also observationally equivalent.

Theorem 10.12. *Observational equivalence is the coarsest consistent congruence.*

Proof. It is a consistent congruence by Lemmas 10.10 and 10.11. Now suppose that R is a consistent congruence, and that $\Gamma \vdash e \; R \; e' : \tau$. To see that $\Gamma \vdash e \cong e' : \tau$, we must show that for all $C : (\Gamma \rhd \tau) \rightsquigarrow (\cdot \rhd \mathsf{ans})$ we have $C\{e\} \simeq C\{e'\}$. But because R is a congruence, we have $\cdot \vdash C\{e\} \; R \; C\{e'\} : \tau$, and because R is consistent, this implies $C\{e\} \simeq C\{e'\}$. □

That said, there are reasons to consider other consistent congruences, e.g. decidability. Discuss "intensional" equality.

Exercise 10.13. Show that $\cdot \vdash \mathsf{yes} \not\cong \mathsf{no} : \mathsf{ans}$.

Exercise 10.14. Show that $x : \mathsf{ans}, y : \mathsf{ans} \vdash x \not\cong y : \mathsf{ans}$.

The problem with observational equivalence is that it is very difficult to establish that two terms *are* observationally equivalent. Imagine trying to show that $\cdot \vdash \mathtt{fst}((\mathsf{yes}, \mathsf{no})) \cong \mathsf{yes} : \mathsf{ans}$. These are obviously Kleene equivalent for $C = \circ$, but even though the left-hand side evaluates in one step to yes, it is not clear how to argue that *every* context will treat them equally. For example, it's not true that every context evaluates its hole, nor is it true that every program containing one of these terms will evaluate to yes. To get a handle on observational equivalence, we will need to turn to—that's right—logical relations.

4 Logical equivalence

The key to analyzing observational equivalence is to consider all the ways in which a program may "use" a subterm $\Gamma \vdash e : \tau$. Many uses are "passive," shuffling e around by pairing it, applying a function to it, etc. These passive uses are equally possible for terms of any type. But then there are the "active" uses of a term, which depend on its type: projecting from a pair, applying a function to an argument, etc. Although passive uses can result in active uses, it turns out (perhaps intuitively) that observational equivalence can be reduced to considering only the active uses.

discuss CIU-equivalence

Definition 10.15 (Logical equivalence)**.** The closed terms $\cdot \vdash e : \tau$ and $\cdot \vdash e' : \tau$ are *logically equivalent at type* τ, or $e \sim e' : \tau$, when:

- $e \sim e' : \mathsf{ans}$ if $e \simeq e'$,
- $e \sim e' : \tau_1 \times \tau_2$ if $\mathtt{fst}(e) \sim \mathtt{fst}(e') : \tau_1$ and $\mathsf{snd}(e) \sim \mathsf{snd}(e') : \tau_2$, and
- $e \sim e' : \tau_1 \to \tau_2$ if for all $e_1 \sim e_1' : \tau_1$ we have $e \; e_1 \sim e' \; e_1' : \tau_2$.

We extend logical equivalence to open terms by quantifying over pointwise logically equivalent closing substitutions.

Definition 10.16. A pair of closing substitutions $\gamma : \Gamma$ and $\gamma' : \Gamma$ are *logically equivalent at context* Γ, or $\gamma \sim \gamma' : \Gamma$, when:

- $\cdot \sim \cdot : \cdot$, and
- $(\gamma, e/x) \sim (\gamma', e'/x) : (\Gamma, x : \tau)$ if $\gamma \sim \gamma' : \Gamma$ and $e \sim e' : \tau$.

Definition 10.17. The terms $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are *(open) logically equivalent*, or $\Gamma \vdash e \sim e' : \tau$, if for all $\gamma \sim \gamma' : \Gamma$ we have $e[\gamma] \sim e'[\gamma'] : \tau$.

We can use a binary logical relations argument to show that open logical equivalence coincides with observational equivalence. (We will see a version of this argument in a future lecture.)

Lemma 10.18 (Head expansion)**.** *If $e \sim e' : \tau$, $d \longmapsto^* e$, and $d' \longmapsto^* e'$, then $d \sim d' : \tau$.*

Theorem 10.19. *Open logical equivalence is the same as observational equivalence: $\Gamma \vdash e \sim e' : \tau$ if and only if $\Gamma \vdash e \cong e' : \tau$.*

Strangely, the fundamental theorem of logical relations here is the statement that logical equivalence is *reflexive*: that for any $\cdot \vdash e : \tau$, we have $e \sim e : \tau$. Or maybe it is not so surprising:

Corollary 10.20 (Termination for observables)**.** *If $\cdot \vdash e : \mathsf{ans}$ then either $e \Downarrow \mathsf{yes}$ or $e \Downarrow \mathsf{no}$.*

Proof. It is easy to see that observational equivalence is reflexive, so $\cdot \vdash e \cong e : \mathsf{ans}$. By Theorem 10.19, $\cdot \vdash e \sim e : \mathsf{ans}$; unfolding the definition of logical equivalence, we have $e \sim e : \mathsf{ans}$ and thus $e \simeq e$, which means that either $e \Downarrow \mathsf{yes}$ or $e \Downarrow \mathsf{no}$. □

Corollary 10.21 (Termination)**.** *If $\cdot \vdash e : \tau$ then $e \Downarrow$.*

Proof. Consider the term $\cdot \vdash (\lambda x : \tau.\mathsf{yes})\ e : \mathsf{ans}$. By Corollary 10.20 we know that $(\lambda x : \tau.\mathsf{yes})\ e \Downarrow$, but by the operational semantics this is only possible if $e \Downarrow$. □

5 Reasoning with logical equivalence

Lemma 10.22. *We have $\cdot \vdash \mathtt{fst}((\mathsf{yes}, \mathsf{no})) \cong \mathsf{yes} : \mathsf{ans}$.*

Proof. It suffices to show $\cdot \vdash \mathtt{fst}((\mathsf{yes}, \mathsf{no})) \sim \mathsf{yes} : \mathsf{ans}$, and hence to show that $\mathtt{fst}((\mathsf{yes}, \mathsf{no})) \simeq \mathsf{yes}$, which is immediate. □

Lemma 10.23 (Head reduction). *If* $d \sim d' : \tau$, $d \longmapsto^* e$, *and* $d' \longmapsto^* e'$, *then* $e \sim e' : \tau$.

Lemma 10.24. *For any* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma \vdash e_2 : \tau_2$, *we have* $\Gamma \vdash \mathtt{fst}((e_1, e_2)) \cong e_1 : \tau_1$ *and* $\Gamma \vdash \mathtt{snd}((e_1, e_2)) \cong e_2 : \tau_2$.

Proof. We focus on the first claim; the second follows similarly. By Theorem 10.19 it suffices to show that for all $\gamma \sim \gamma' : \Gamma$, $\mathtt{fst}((e_1[\gamma], e_2[\gamma])) \sim e_1[\gamma'] : \tau_1$. By termination and substitution we have $e_1[\gamma] \Downarrow v_1$, $e_1[\gamma'] \Downarrow v_1'$, and $e_2[\gamma] \Downarrow v_2$; by the operational semantics, we have $\mathtt{fst}((e_1[\gamma], e_2[\gamma])) \Downarrow v_1$ and $e_1[\gamma'] \Downarrow v_1'$.

By the reflexivity of open logical equivalence, we have $e_1[\gamma] \sim e_1[\gamma'] : \tau_1$, and by head reduction we have $v_1 \sim v_1' : \tau_1$. The result follows by head expansion. □

Lemma 10.25. *For any* $\Gamma \vdash e : \tau_1 \times \tau_2$, *we have* $\Gamma \vdash e \cong (\mathtt{fst}(e), \mathtt{snd}(e)) : \tau_1 \times \tau_2$.

Proof. By Theorem 10.19 twice, it suffices to show that for all $\gamma \sim \gamma' : \Gamma$,

$$\cdot \vdash \mathtt{fst}(e[\gamma]) \cong \mathtt{fst}((\mathtt{fst}(e[\gamma']), \mathtt{snd}(e[\gamma']))) : \tau_1$$
$$\cdot \vdash \mathtt{snd}(e[\gamma]) \cong \mathtt{snd}((\mathtt{fst}(e[\gamma']), \mathtt{snd}(e[\gamma']))) : \tau_2$$

By the previous lemma, $\cdot \vdash \mathtt{fst}((\mathtt{fst}(e[\gamma']), \mathtt{snd}(e[\gamma']))) \cong \mathtt{fst}(e[\gamma']) : \tau_1$; by transitivity it suffices to show that $\cdot \vdash \mathtt{fst}(e[\gamma]) \cong \mathtt{fst}(e[\gamma']) : \tau_1$, which follows from reflexivity of open logical equivalence. The other case is similar. □

extensional equality; beta and eta for function types; counting functions

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[Mor69] James Hiram Morris Jr. "Lambda-calculus models of programming languages". PhD thesis. Massachusetts Institute of Technology, 1969. URL: http://hdl.handle.net/1721.1/64850.