Lecture Notes 10
Observational Equivalence

Carlo Angiuli

B522: PL Foundations
April 6, 2026

We now return to the question of when two programs 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, such as $\lambda x : \mathtt{unit}.x$ and $\lambda x : \mathtt{unit}.()$ at type $\mathtt{unit} \to \mathtt{unit}$, or $\lambda x : \tau.\mathtt{fst}((x, x))$, $\lambda x : \tau.\mathtt{snd}(((), x))$, and $\lambda x : \tau.x$ at type $\tau \to \tau$; we called these *extensionally equal* functions in the lecture on type isomorphism. In addition, this relation does not tell us anything about the equality of *open* terms; we might imagine that the terms $x + y$ and $y + x$ "should" be equal in context $x : \mathtt{nat}, y : \mathtt{nat}$.

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

Before precisely defining observational equivalence, it is worth exploring what properties we expect it to satisfy. As it turns out, the foregoing discussion already pins things down considerably.

First of all, any sensible notion of "equality" should satisfy certain conditions:

- It should be a *binary relation*: a property of a pair of things (here, terms).

- 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-forming operations. For example, if $e = e'$ then we should also have $\texttt{fst}(e) = \texttt{fst}(e')$, $f\ e = f\ e'$, $\lambda x : \tau.e = \lambda x : \tau.e'$, etc.

These properties allow us to chain together equalities and "replace equals by equals" anywhere inside a larger term; in short, they allow us to treat equal terms as genuinely interchangeable. In particular, transitivity and congruence combine to give us familiar kinds of equational reasoning: if $f = f'$ and $e = e'$ then we have $f\ e = f'\ e'$ by $f\ e = f\ e' = f'\ e'$.

The last instance of congruence above—that two λs should be equal when their bodies are equal—implies that

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

Because we are studying type systems, it makes sense to only consider equalities between terms that are well-typed, and *a fortiori*, that:

- Equality should be defined only for pairs of terms of the *same type in the same context*.

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. Perhaps it is more correct to say that we are defining a family of equivalence relations, one at each pair of Γ and τ, where these equivalence relations are connected to one another via congruence conditions. For example, if two terms are equal at $\Gamma, \tau_1 \times \tau_2$, then their first projections must be equal at Γ, τ_1.

We have discussed general properties of equality, and specific properties of equality connected to type systems. What about operational semantics?

- 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 we must define equality for open terms as well.

The following properties are less mandatory from a theoretical perspective, but very helpful in practice:

- Equality at function types should contain extensional equality.
- Equality of open terms should contain β-equivalence; more generally, it should extend all the principal reductions to open terms. For example, if $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash e_2 : \tau_2$ then we should have $\Gamma \vdash \texttt{fst}((e_1, e_2)) = e_1 : \tau_1$, rather than only having this equation when $\Gamma = \cdot$.
- Equality should not be the total relation; that is, there should be at least one equality that could hold but doesn't.

That's a lot of criteria, and it is not at all obvious whether zero, one, ten, or infinitely many relations satisfy all of them. Our final criterion, a sort of "meta-criterion," is that from all of the relations satisfying these criteria we should choose a "canonical" one, or one that is somehow memorable.

- We should choose the "maximal" equality relation (i.e., the one relating the most terms) satisfying all of the above properties.

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 whole idea that functions evaluate to λ values is a theoretical simplification of structural operational semantics that does not reflect any realistic implementations. 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 practice, however, 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.

where to discuss PCF and parallel `or`?

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 with finite products, or at least a variation on it. We cannot use that language directly 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 with finite products, 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:

Types	$\tau ::=$	$\vdots$	$\vdots$	$\vdots$
		`ans`	`ans`	answer type
Terms	$e ::=$	$\vdots$	$\vdots$	$\vdots$
		`yes`	`yes`	yes answer
		`no`	`no`	no answer

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

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

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:

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

An example of a term context is $\mathsf{fst}(\mathsf{yes}, (\lambda x : \mathsf{ans}.(x, \circ)))$. 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:

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

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 $\cdot \vdash e : \mathsf{ans}$ and $\cdot \vdash e' : \mathsf{ans}$, we say that e and e' are *Kleene equivalent*, written $e \simeq e'$, when one of the following (mutually exclusive) conditions holds:

- $e \Downarrow \mathsf{yes}$ and $e' \Downarrow \mathsf{yes}$, or
- $e \Downarrow \mathsf{no}$ and $e' \Downarrow \mathsf{no}$, or
- $e \not\Downarrow$ and $e' \not\Downarrow$ (i.e., there exists no v such that $e \Downarrow v$, and likewise for e').

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.

Remark 10.9. If STLC++ terminates, then $e \simeq e'$ if and only if e and e' both evaluate to yes or both evaluate to no. And as one might expect, STLC++ *does* terminate, which one can show by a trivial modification of our termination proof for STLC. So why does Definition 10.8 include a nontermination clause?

It is very important to the theory of observational equivalence that Kleene equivalence be reflexive, but reflexivity of the "simpler" definition is tantamount to termination: for all $\cdot \vdash e : \mathsf{ans}$, either $e \Downarrow \mathsf{yes}$ or $e \Downarrow \mathsf{no}$. Therefore, using the simpler definition forces us to prove termination before defining observational equivalence, which is a rather long detour—assuming that it holds at all!

Note that Chapters 46 and 48 of Harper [Har16], which characterize observational equivalence of System T and System F respectively, erroneously claim that the simpler definition is reflexive without remarking on termination. In fact, Chapter 46 claims termination as a corollary of the fundamental theorem of logical relations for logical equivalence (Corollary 46.15), when in fact termination is required (via reflexivity of Kleene equivalence) in the proof of that theorem.

Definition 10.10 (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 $C : (\Gamma \rhd \tau) \rightsquigarrow (\cdot \rhd \mathsf{ans})$ 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 10.11. *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 10.12. *Observational equivalence is* consistent *with Kleene equivalence: if* $\cdot \vdash e \cong e' : \mathsf{ans}$ *then* $e \simeq e'$.

Importantly, Lemma 10.12 implies that observational equivalence does not equate all terms: if it did then it would in particular 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 10.13. *Observational equivalence is the coarsest consistent congruence.*

Proof. It is a consistent congruence by Lemmas 10.11 and 10.12. 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 $C\{e\} \simeq C\{e'\}$. □

Exercise 10.14. Show that $x : \mathsf{ans}, y : \mathsf{ans} \vdash x \ncong y : \mathsf{ans}$.

The problem with observational equivalence is that it is very difficult to establish that two terms *are* observationally equivalent. Imagine trying to show that $\cdot \vdash \mathtt{fst}((\mathtt{yes}, \mathtt{no})) \cong \mathtt{yes} : \mathsf{ans}$. These are obviously Kleene equivalent for $C = \circ$, but even though the left-hand side evaluates in one step to $\mathtt{yes}$, 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 $\mathtt{yes}$. To get a handle on observational equivalence, we will need to turn to—that's right—logical relations.

Remark 10.15. One may reasonably wonder if there are any interesting examples of consistent congruences besides the coarsest one, and why anyone might choose one of those over observational equivalence. To answer the second question first, one of the main reasons is precisely that observational equivalence is so intractable: even armed with logical relations, it is in most cases undecidable. An example of a less powerful but more tractable consistent congruence is β-equivalence, which does not contain extensional equality but satisfies most of our other criteria.

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.

Definition 10.16 (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 10.17. 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 10.18. 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.) 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 $e \sim e : \tau$.

Lemma 10.19 (Head expansion)**.** *If* $e \sim e' : \tau$, $d \longmapsto^* e$, *and* $d' \longmapsto^* e'$, *then* $d \sim d' : \tau$.

Theorem 10.20. *Open logical equivalence is the same as observational equivalence:* $\Gamma \vdash e \sim e' : \tau$ *if and only if* $\Gamma \vdash e \cong e' : \tau$.

sketch other main ideas of proof?

Remark 10.21. For another take on the idea that observational equivalence can be reduced to considering the "active" uses of terms, there is a *third* equivalent notion of equivalence that is halfway between observational equivalence, which quantifies over all program contexts, and logical equivalence, which is completely type-directed. This notion is known as *ciu-equivalence*, short for *closed instantiations of uses*, and was introduced by Mason and Talcott [MT91].

Two terms $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are *ciu-equivalent* if for every closing substitution $\gamma : \Gamma$ and for every *evaluation context* $\mathcal{E} : (\cdot \rhd \tau) \rightsquigarrow (\cdot \rhd \mathsf{ans})$, we have $\mathcal{E}\{e[\gamma]\} \simeq \mathcal{E}\{e'[\gamma]\}$. (Note that evaluation contexts never descend under binders, so we must close the terms before putting them in an evaluation context.) Observational equivalence straightforwardly implies ciu-equivalence, and with additional work they can be shown to be equivalent.

5 Reasoning with logical equivalence

We close by briefly sketching how the above definitions and lemmas can be used to establish interesting observational equivalences.

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

Proof. It suffices to show $\cdot \vdash \mathtt{fst}((\mathtt{yes}, \mathtt{no})) \sim \mathtt{yes} : \mathsf{ans}$, and hence to show that $\mathtt{fst}((\mathtt{yes}, \mathtt{no})) \simeq \mathtt{yes}$, 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* $\Gamma \vdash \mathsf{snd}((e_1, e_2)) \cong e_2 : \tau_2$.

Proof. We focus on the first claim; the second follows similarly. By Theorem 10.20 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* $\Gamma \vdash e \cong (\mathtt{fst}(e), \mathsf{snd}(e)) : \tau_1 \times \tau_2$.

Proof. By Theorem 10.20 twice, it suffices to show that for all $\gamma \sim \gamma' : \Gamma$,

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

By the previous lemma, $\cdot \vdash \mathtt{fst}((\mathtt{fst}(e[\gamma']), \mathsf{snd}(e[\gamma']))) \cong \mathtt{fst}(e[\gamma']) : \tau_1$; 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.

[MT91] Ian Mason and Carolyn Talcott. “Equivalence in functional languages with effects”. In: *Journal of Functional Programming* 1.3 (1991), pp. 287–327. DOI: 10.1017/S0956796800000125.