Comp Sci 704
Fall 2024

Lecture 15: Observational Equivalence

Ethan Cecchetti

How do we know when two λ-calculus terms mean the same thing? The question is not as simple as it may seem. We could demand that the two terms be syntactically equal—the strictest plausible definition—but that is not very interesting or very useful. For example, $\lambda x.\, x$ and $\lambda y.\, y$ certainly represent the same function, as the argument name fundamentally does not matter. This example suggests we could declare two terms equivalent if they are α-equivalent. This definition is reasonable if one regards λ-terms as *intensional* objects.

If we wish to view them *extensionally*, however, this definition is far too strict. Ideally, we would like to consider two terms equal if they represent the same function. Certainly $\lambda x.\, x$ and $\lambda y.\, y$ represent the same function, but they (and other α-equivalent terms) are not the only representations of the identity function. For instance $\lambda x.\, (\lambda y.\, y)\ x$ is also the identity function.

Note that $\lambda x.\, (\lambda y.\, y)\ x$ reduces to $\lambda x.\, x$ in one β-reduction step. We might declare two terms equal if either (i) they have a common normal form up to α-equivalence, or (ii) neither has a normal form. That is, either they both converge to α-equivalent values under some sequence of reductions, or neither converges under any sequence of reductions. This is sometimes known as *β-equivalence*, written $e_1 =_\beta e_2$. By confluence, β-equivalence is an equivalence relation. It is useful for compiler optimization and for checking type equality in some advanced type systems. Unfortunately, it would not work for reduction strategies like CBN and CBV, which do not allow reductions inside the bodies of λ-abstractions, and therefore β-equivalence is just α-equivalence on λ-abstractions.

So how, then, do we know if two terms behave the same way? It would be nice if we could just say that two terms are equivalent if they give equivalent results on equivalent inputs. Unfortunately, this is a circular statement, so it doesn't define anything! It is not even clear that there is a "right" definition.

Another complication is undecidability. It is likely that any reasonable notion of extensional equivalence will be undecidable due to the relationship between the λ-calculus and Turing machines. If we could test equivalence, then we could test equivalence with Ω, which is tantamount to solving the halting problem.

1 Observational Equivalence

To get a truly extensional view of equivalence, we want to say that two λ-terms are the same if they behave the same way in every possible context. But what do we mean by "behave the same" and how do we define a "context"?

For simplicity, let us work with only closed λ-terms, so nothing gets stuck, and assume an evaluation strategy such as CBV or CBN that is *deterministic*, which means that there is at most one next β-reduction that can be performed. We say that a term *e terminates* or *converges* if there is a finite sequence of reductions

$$e \longrightarrow e' \longrightarrow e'' \longrightarrow \cdots \longrightarrow v$$

where v is a value. We write $e \Downarrow v$ when this happens, or $e\Downarrow$ to mean there is some v such that $e \Downarrow v$.

The other possibility is that it keeps on reducing forever without ever arriving at a value. When this happens, we say that *e diverges* and write $e\Uparrow$. Because we have assumed that we are using a deterministic evaluation strategy, exactly one of these two cases will occur.

With CBN or CBV, there are infinitely many divergent terms. One example is Ω. We might consider all divergent terms equivalent, since none of them produce a value.

Now we get back to the question of defining "behave the same" and "in any context." A *context* is any term $C[\cdot]$ with a single occurrence of a distinguished special variable, called the *hole*, and $C[e]$ denotes replacing the hole with the expression e. These work the same way as the evaluation contexts discussed in

Lecture 13. The BNF grammar for a context is:

$$C \;::=\; [\cdot] \mid C\,e \mid e\,C \mid \lambda x.\,C.$$

To formalize the notion of two programs behaving the same way, we want to say that, when put into the same contexts, they must produce the same outputs. That requires a way to compare output values.

For now, let us assume that we have some equivalence relation $\equiv$ on values. We can construct a notion of *observational equivalence* by saying that two programs e_1 and e_2 are observationally equivalent if, in every context, they either both diverge or produce values that are equivalent according to $\equiv$. In mathematical notation,

$$\begin{aligned} e_1 \equiv_{\text{obs}} e_2 \overset{\triangle}{\iff} \forall C.\ &\text{either } (C[e_1]\Uparrow \wedge C[e_2]\Uparrow) \\ &\text{or } (C[e_1] \Downarrow v_1 \wedge C[e_2] \Downarrow v_2 \wedge v_1 \equiv v_2). \end{aligned}$$

In other words, for every context C, either $C[e_1]$ and $C[e_2]$ both diverge, or both converge and produce $\equiv$-equivalent terms.

Note that on values themselves, equivalence is not necessarily the same as observational equivalence. Any two values v_1 and v_2 that are observationally equivalent are certainly $\equiv$-equivalent, though. Simply take $C = [\cdot]$ and $C[v_1] \Downarrow v_1$ and $C[v_2] \Downarrow v_2$, so if $v_1 \equiv_{\text{obs}} v_2$, then $v_1 \equiv v_2$. The converse, however, is not necessarily true. As an extreme example, if $\equiv$ is the maximal equivalence relation, relating every value to every other value, then $\lambda x.\,x\,x \equiv \lambda x.\,x$, but those are certainly not observationally equivalent (consider the context $C = (\lambda x.\,x\,x)\ [\cdot]$).

Is it possible to have the two coincide on values? In other words, is there a fixed point of the transformation $\equiv\ \mapsto\ \equiv_{\text{obs}}$? If so, is it unique? Even if not, is there a reasonable choice for the definition of extensional equivalence?

The answers to these questions lie in the following facts, none of which are difficult to prove.

1. If $\equiv$ is an equivalence relation, then $\equiv_{\text{obs}}$ is an equivalence relation.
2. Restricted to values, $\equiv_{\text{obs}}$ refines $\equiv$ on values. That is, viewed as sets of ordered pairs restricted to values, $\equiv_{\text{obs}} \subseteq\ \equiv$. We just proved this above.
3. The transformation $\equiv\ \mapsto\ \equiv_{\text{obs}}$ is *monotone* with respect to the refinement relation. That is, if $\equiv^1\ \subseteq\ \equiv^2$, then $\equiv^1_{\text{obs}}\ \subseteq\ \equiv^2_{\text{obs}}$.

These three facts together are sufficient to prove that there is, indeed, a fixed point.

It turns out that there can be several fixed points of the transformation $\equiv\ \mapsto\ \equiv_{\text{obs}}$ depending on the reduction strategy. For instance, for CBN and CBV, syntactic equality is a fixed point (in fact, it is the smallest fixed point), as is α-equivalence and β-equivalence (which coincide on values).

To get as extensional a definition of equivalence as possible, we want the equivalence that relates the most programs without equating observationally distinct programs. To get that, we want the *coarsest* (or largest) fixed point, which is refined by every other fixed point. For CBN and CBV, that fixed point is as follows:

$$e_1 \equiv_{\Downarrow} e_2 \overset{\triangle}{\iff} \forall C.\ C[e_1]\Downarrow \text{ iff } C[e_2]\Downarrow.$$

Theorem 1. *For CBV and CBN, the relation $\equiv_{\Downarrow}$ is a fixed point of $\equiv\ \mapsto\ \equiv_{\text{obs}}$—that is, $\equiv_{\Downarrow}\ =\ (\equiv_{\Downarrow})_{\text{obs}}$. Moreover, it is the coarsest such fixed point.*

The relation $\equiv_{\Downarrow}$ is a good candidate for extensional equivalence. By definition, e_1 and e_2 are observationally equivalent if they have the same convergence behavior in every context. It is unnecessary to compare the resulting values when both converge. The intuition behind not needing to check is that, if there was a way to observe a difference between the values, there would be a larger context that ran the initial computation and then converged on one output and diverged on the other. Thus the quantification over all contexts captures the idea that the output values must be extensionally equivalent as well.

This definition of observational equivalence importantly depends highly on the features of a language. For instance, in a language with primitive numbers but no way to distinguish them—no comparison operators or equality checks—all numbers are observationally equivalent to all other numbers. However, if we add a simple equality check or comparison operator, they all become observationally distinct.

In the most extreme case, in a language like Java with fully reflection—a feature that allows a program to observe its own source code at run time—observational equivalence devolves into syntactic equality. Since a program can observe its own source code, one can construct a context C that observes the source code of whatever expression is put in the hole, compares that to the source code for e, and terminates if they are the same and not if they are different. That context will distinguish e from any syntactically distinct program, reducing observational equivalence to syntactic equality. Most languages do not support full reflection, however, making observational equivalence a powerful and interesting notion.